INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports Techniques

N° 123

Programme 4
Bases de Données

THE RDL/C LANGUAGE REFERENCE MANUAL V1

G. KIERNAN
C. de MAINDREVILLE

Octobre 1990

The RDL/C Language
Reference Manual V1

Gerald Kiernan and Christophe de Maindreville
I.N.R.I.A. Rocquencourt
78135 Le Chesnay, FRANCE
INFOSYS
15, rue A.France
92800 Puteaux, France

Abstract

This manual describes the usage of a rule language compiler for a relational DBMS. The rule language described in this manual is called RDL/C. It is based on a production rule language and on C code. RDL/C includes a rule section to define in a declarative way, general knowledge about the data stored in the database. This language includes procedural constructs such as explicit control over the rules and calls to a language such as C. Rules and procedures exchange parameters in an easy way. The compiler is designed as an application generator which translates rule programs into C based database applications.

Le Langage RDL/C [1]
Manuel de Référence V1

Résumé:

Ce rapport décrit un langage de règles de production pour base de données relationnelles. Le langage RDL/C supporte une programmation déclarative basé sur un langage de règles de production et une programmation procédurale qui utilise le langage C. Le langage RDL/C étant orienté base de données, les programmes sont exécutables par le SGBD sans chargement de données dans la mémoire de C. Le langage de règles et le langage C communiquent à travers des variables C qui peuvent apparaitre dans les parties action et condition d'une règle. Le compilateur traduit des programmes RDL/C en programmes C qui contiennnent des requètes SQL.

---

[1] Ce travail a été partiellement financé par le Ministère de la Recherche et des Technologies

Contents

1. Preface

This manual describes the usage of a rule language compiler for a relational DBMS. The rule language described in this manual is called RDL/C. It is based on a production rule language called RDL1 and on C code. The compiler is designed as an application generator which translates rule programs into C based database applications. The DBMS system that is described in this manual is the SABRINA DBMS [Sabrina1]. SABRINA supports the standard features of a relational DBMS in addition to an Abstract Data Type (ADT) facility. This ADT facility is accessible from within the production rules of RDL/C.

The introduction presents the general ideas behind the research in the area of deductive DBMS. The Data Model section describes the relational data model of the SABRINA DBMS. This section introduces the notion of Abstract Data Type and assumes that the user is familiar with relational systems. The next section is a brief overview of the language introducing the main ideas of the language. This section is intended for users who are not familiar with production rule environments for database. The general notions of deductive DBMS are presented in this section.

The section on the language is the heart of the manual. This section details all aspects of the language. Users writing deductive DBMS applications can refer to this section to write rule modules and fix compile-time errors. The terms of the language are described. These are constants, attributes, functions and main memory variables. Terms appear in expressions which are also described in this section. The notion of C program code appearing in rule modules is introduced. The production rule is described next. The production rule is the key element of the language. Then the control string is presented in the next section. The control string induces a partial ordering among rules. This section also outlines all parts of a rule module. These are: include statements, the module name, the variable declaration section, the relation declaration section, the report section, the production rule section, the control string, the initialization code section, the wrapup code section, and the end module statement.

A sample database is introduced in the following section. This sample is a geographic information system based on a geographic data model designed by researchers from the National Geographic Institute (IGN) in France in the framework of the ESPRIT Project TROPICS. Readers may find that this example is too complicated. However, the usefulness of RDL/C cannot be demonstrated on a toy database where a language like SQL is entirely sufficient. The geographic database has the quality of being too complicated for SQL and therefore, of being adequate for a language like RDL/C. The geographic data model is supported by the relational model with one abstract data type defined to support the geometric quality of geographic data. A sample application is then described. The application is based on a shortest path algorithm. A map of the region of Toulouse in the south of France is displayed. The user is asked to select a district where there is a fire. Then he is asked to select the departure district and the arrival district. These districts are first displayed

to the user using X windows facilities. Then, the program searches for a path along the roads in the database avoiding roads going through the district where there is a fire.

The next section looks at the problem of compiling rule programs and details the various compilation options. Running rule applications is then discussed. Two important header files are described. These are to be included in all C programs that interact with rule modules. The two files are the iml.h file and the rdlHeader.h file. The iml file has declarations necessary to establish contact with the DBMS and to query the database with SQL. The rdlHeader file has the declarations necessary to communicate with rule modules. The conclusion of the manual is given. Two annexes follow the conclusion. The first describes the BNF of the rule language. This can be useful in the case of hard to overcome syntax errors. The last section describes the main types of error messages that can occur.

2. Introduction

The RDL/C approach can provide a rule interface on top of any relational DBMS. More precisely, the approach taken for RDL/C provides :

(1) a rule language suitable for data-intensive applications. This language is more suitable than C-SQL for traditional applications and more adequate than PROLOG for *intelligent data-intensive* applications. RDL/C includes a rule section to define in a declarative way, general knowledge about the data stored in the database. This language includes procedural constructs such as explicit control over the rules and calls to a language such as C. Rules and procedures exchange parameters in an easy way. However, the initial semantics of the rule language remains declarative and easily comprehensible to the user.

(2) programs which run on top of any relational DBMS which supports ADT facilities. The interface between the rule language and the DBMS is SQL. The compiler produces C/SQL code which runs over the DBMS.

(3) efficiency, and standard database system features. The first requirement is achieved by the fact that all data are manipulated by the DBMS. The second feature is achieved by the high-level interface between the rule language and the DBMS, i.e, SQL.

The RDL/C language is a rule based language for querying and updating database relations. The RDL/C data model is an extension of the relational model with the support of Abstract Data Types (ADT). The RDL/C language is derived from the RDL1 language defined in [Maindreville88]. It supports limited data functions, negation, quantifiers and updates. It has been extended to support an external control of rule programs by the use of a language of control. Communication with an external language is done through user defined data functions which can appear in the calculus or in procedural sections attached to the rules.

The main motivation which lead to design and implement such a language is the following one :

> The Datalog language and its extensions do not seem appropriate for the development of real deductive database applications. It does not provide complete programming language capabilities such as control structures, main memory variables, procedure calls and side effects, interaction with the user. It is also more query oriented than application oriented. RDL/C extends Datalog with the support for these programming features. However, the underlying data model stays relational.

The RDL/C compiler accepts a source program and produces as output, a C program which implements the rule program. The DBMS does not require any inferencing capabilities to process the program. The C program contains code to implement each rule and includes the inference engine which fires rules until a fixpoint is reached. All data remains in the DBMS during the inference process. This is because rules are based on relational calculus and can thus be solved by the DBMS.

3. Data Model

3.1. Abstract Data Types

The support of Abstract Data Types (ADT) provides a rich typing capability for relational database systems. An ADT is a type described by its operational semantics, i.e., by a set of operations which can be performed on the instances of that type. For example, the type Stack is described by the push, pop and top operations and not by the data structure that implements a stack. Types in languages like Pascal and C are described structurally.

The ADT capability is supported by User-Defined Data Types (UDT) and User-Defined Functions (UDF). UDTs generalize the notion of domain in the relational model. Thus, a domain is defined either as a basic data type (real, integer, boolean, string) or as a user-defined data type built from basic data types and type constructors that depend on the UDT implementation language. Previously defined UDTs can be used in turn to build new data types. From the input languages (extended SQL and RDL/C), an ADT is viewed as a set of (user-defined) functions, the UDF, that operate on instances of the defined type.

The UDT/UDF implementation language supported by the system is LISP [Sabrina2]. From the user point of view, LISP creates and manipulates complex hierarchical structures required by applications. An interpreted environment protects the DBMS from abnormal termination in case of programming errors.

3.2. User-defined Data Types

A UDT is specified using a functional notation. The name of the data type is also the name of a Boolean function that evaluates to True if its parameter qualifies as an instance of the defined type. UDTs can be recursively defined using basic data types and existing UDT. Furthermore, UDT are described in a ISA hierarchy capturing the usual notion of type inheritance. Thus, UDT operators can be inherited along the hierarchy. The extension of the set of basic types together with the UDTs form the set of domains from which tuples and relations can be built.

The specification of a new type requires the use of the **dd** primitive. It specifies the name of a super-type if it is a specialization of another type. The type specification is the code used to check whether a datum is an instance of the type. A UDT type specification is a LISP function that returns either a True or a Nil value. Its syntax is:

```
(dd <super-type name>:<type name>
(<type specification>))
```

Examples :

The "set" domain can be defined as follows:

```
(dd #:t:set (x)
(cond
      ((null x) t)
      ((member (car x)(cdr x)) nil)
      (t (apply '#:t:set (cdr x)))))
```

The support for UDT are the basic DBMS domains which are integers, real numbers and text strings. Text strings are used to support complex data structures as well as plain text. The domain "point" can be defined as follows:

```
(dd #:t:point (x)
  (and
    (numberp (car x))
    (numberp (car (cdr x)))
    (null (cdr (cdr x)))))
```

The domain "listofPoints" is defined as below:

```
(dd #:t:listofPoints (x)
  (cond
    ((null x) t)
    ((and
      (#:t:point (car x))
      (apply '#:t:listeDePoints (cdr x)) t)
    (t nil)))
```

The domain "polygon" is defined as a specialization of "listofPoints":

```
(dd #:t:listofPoints:polygon (x)
     (and (apply ':listofPoints x)
          (equalPoint (car x) (last x))))
```

Then, a new relation called "map" can be created with a reference to this domain:

```
CREATE TABLE map (mapid integer, contour polygon, ...)
```

3.3. User-defined Functions

User-defined functions (UDF) are built from a library of primitive functions. These include standard LISP functions and window management functions (a C package) which allow the user to implement graphics and user interaction. The specification of a new function over UDT requires the use of the **de** primitive. It specifies the name of the function, the type of its arguments, and the type of its result. The syntax of the command is:

```
(de  <function-name> (par1, ...parn)
     <function-body>)
```

Examples :

One could define the function card which returns the cardinality of a set :

```
(de #:(#:set):card (x)
     (cond
```

```
((null x) 0)
(t (+ 1 (:card (cdr x))))))
```

The functions are inherited from a type to its subtypes. A UDF can also be redefined on a subtype (overloading). For instance, the surface function defined on polygon can be redefined for triangle. Function selection is done according to the type of all the arguments of the function. The function selection mechanism can be operationally defined as follows:

```
F (P1, ..., Pn)
    where
        F is a function name, and
        each Pi is either
            - a constant,
            - an attribute,
            - a function
            and is of type Ti
```

Function selection considers all parameters (multi-targetting):

```
F (Ti, ..., Tn)
    where
        each Ti is a path name in the hierarchy.
```

The problem with multi-targetting is insuring that only one function qualifies the selection mechanism. This problem is resolved operationally by varying i+1 faster than i. The following example illustrates the mechanism.

Select a function for the following expression :

```
F(p, q)
```

where both p and q are of type #:A:B:C. If there is one function defined as #:(#:A #:A):F, the system will find it by going through the hierarchy as can be seen below.

```
#:(#:A:B:C #:A:B:C):F
#:(#:A:B:C #:A:B:):F
#:(#:A:B:C #:A:):F
#:(#:A:B: #:A:B:C):F
#:(#:A:B #:A:B:):F
#:(#:A:B #:A):F
#:(#:A #:A:B:C):F
#:(#:A #:A:B):F
#:(#:A #:A):F
```

3.4. Access to the Lisp Environment

The LISP environment is reachable from the SQL environment. To define new complex domains or new functions, the user types the LISP command. Then the prompt ? is displayed on the screen. The user can define and test his operators under the LISP environment. After this definition phase, the new domains can be stored into the DBMS using the following command :

```
(save <function name> <result type>)
```

Examples :

```
(save #:t:listofPoints:polygon )
```

stores the complex domain polygon

```
(save #:(#:t:listofPoints:polygon ):surface r)
```

stores the definition of user defined function surface. The type of the result is r meaning real.

3.4.1. Example of a working session.

The following example illustrates the steps a user would go through to define the ADT function CARDinality for polygons. The cardinality is the number of points in the polygon. The '>' character is the SQL prompt character which indicates that the system is ready to accept SQL statements. The LISP command is an SQL command which puts the user in the ADT definition environment. The '?' is the LISP prompt character which indicates that the interpreter is ready to process ADT definitions. What follows the '=' character is the interpreters response to the user's input.

```
> select *
> from rectangles ;
...
> lisp;
*** SABRINA-LISP (exit with END)
?
? (de #:(#:t:listofPoints:polygon):card (x)
?   (cond
?      ((null x) 0)
?      (t (add1 (#:(#:t:listofPoints:polygon):card (cdr x))))))
= #:(#:t:listofPoints:polygon):card
?
? (#:(#:t:listofPoints:polygon):card '((4 5)(4 55)(45 55)(4 5)))
= 4
? (save #:(#:t:listofPoints:polygon):card integer)
= #:(#:t:listofPoints:polygon):card
? end
>
> select r.*, card (sides)
> from rectangles as r;
```

The de function allows to define new ADT functions. The Card ADT is defined for polygons. The interpreter responds by accepting the definition of the function simply by returning its name. Then, the user tests the Card function on sample data. The card function returns the value 4 after having evaluated the sample data. Then, the definition is saved in the DBMS using the Save function. The result type of the card function is specified as the last argument to the Save function. Card returns an integer result. End signals the end of the ADT definition cycle. The user returns to SQL and can

immediately test the Card operator in an SQL select statement. This SQL statement selects all tuples in the Rectangles relation and calculates the cardinality of the sides attribute for all tuples.

3.5. Using ADT in SQL Statements

3.5.1. Complex Object Creation

The relational DBMS implements an SQL interface which is based on the SQL norm. The language has been extended to manipulate ADT. SQL++ is the name of this extended language interface [Sabrina2]. The database administrator extends the DBMS by defining new ADT which are then made available for use in the external language. In this section, the various extensions brought to SQL to include ADT are considered.

Relations are created using the CREATE TABLE command. For example, the RECTANGLES relation is created in the following:

```
CREATE TABLE RECTANGLES   (
      R#          integer,
      COLOR       text,
      SIDES       polygon) ;
```

This relation contains three attributes where the first two are of standard domains and the last one is a complex domain. The SIDES attribute takes its values from the polygon domain which has been defined in the previous section. Once the rectangles relation has been created, values may be inserted into the relation using the INSERT command. For example,

```
INSERT INTO RECTANGLES VALUES (1, BLEU, ((4 5) (4 55) (45 55) (4 5))) ;
```

When new values are inserted into relations, the ADT function which implements domain integrity constraints validation are run over the new values to determine if the values qualify as occurrences of the domain. The same check applies when ADT values are updated. Here, the value ((4 5)...) qualifies as an occurrence of the polygon domain. Note that the system does not implement object identity. However, it could be possible to identify complex objects within the validation function; that would make possible referential sharing among complex objects (i.e., complex domain values).

3.5.2. Complex Object Selection

A ADT function can be used in any clause of a relational expression (projection, restriction, aggregation, sort) and is applicable to one or more attributes. A ADT function F applied to a number n of arguments is written as F(P1..Pn). The parameters Pi can be constants, attributes or ADT functions applied to other parameters. The F function will be selected according to all the parameter types. Functions may also appear according to their complete name (as in their declaration) and thereby bypassing the inheritance based selection mechanism. The following examples demonstrate the various possibilities:

Example 1 : A ADT function appears in the projection clause. The query selects all attributes in the relation in addition to the surface value of the sides attribute.

```
SELECT    *,  SURFACE (SIDES)
FROM      RECTANGLES ;
```

Example 2 : ADT functions are used in a restriction clause. This query selects those rectangles with a height greater than their width.

```
SELECT    *
FROM      RECTANGLES
WHERE     HEIGHT (SIDES) > WIDTH (SIDES) ;
```

Example 3 : A ADT function is used in a join expression. This query selects rectangles with different surface values and displays the greatest of the two values.

```
SELECT    *, BIG (R1.SIDES, R2.SIDES)
FROM      RECTANGLES AS R1, RECTANGLES AS R2
WHERE     SURFACE (R1.SIDES) <> SURFACE (R2.SIDES) ;
```

Example 4 : A ADT function is used to sort a relation. This query sorts the tuples of the Rectangles relation in ascending order of cardinality of the sides attribute.

```
SELECT    R.NR, CARD (SIDES)
FROM      RECTANGLES AS R
ORDER     2 ASC;
```

4. Overview of the Language

RDL/C is derived from RDL1 [Maindreville88, Kiernan90], a production rule language which has been integrated in the Sabrina RDBMS [Sabrina1]. The RDL/C language supports declarative programming based on RDL1 and procedural programming based on C code. In this section, we present an overview of the language through some examples. The complete syntax and semantics of RDL/C are detailed in the next section.

4.1. Illustrative Examples

The purpose of these examples is to show the main features of the language: The ability to query a database, the declaration of an intentional database through production rules, the integration of C statements which manipulate main memory variables, and the interaction with the user. Let us consider a base relation Person having for schema Person (name char, age integer).

The first rule program inserts into the deduced relation Same-Age all the Persons who are have the same age :

```
MODULE M0 ;
BASE
            Person (name char, age integer);
DEDUCED
            Same_Age (name1 char, name2 char) ;

RULES
r1 is
      IF Person(x) and Person(y) (x.age = y.age)
            THEN + Same_Age (name1 = x.name, name2 = y.name) ;
END MODULE
```

The following rule program asks the user to enter a value for the C-variable age and then, fires the rule r1 using the current value of age. This illustrates the use of main memory variables in rules and interaction with the user.

```
MODULE M1 ;
BASE
            Person (name char, age integer);
DEDUCED
            OldPerson like Person;
VAR
            integer age;
RULES
r1 is
      IF Person(x) ( x.age >= age)
            THEN + OldPerson(x) ;
INIT
            {scanf("%d", age);}
END MODULE
```

Let us consider the slightly modified module :

```
MODULE M2 ;
BASE
            Person (name char, age integer);
DEDUCED
            OldPerson like Person;
VAR
            integer age;
RULES
```

```
r1 is
      {scanf("%d", age);}
      IF Person(x)  ( x.age >= age)
            THEN + OldPerson(x) ;
                  {printf ("Successful firing with %d\n", age);}
END MODULE
```

This module asks the user an initial value for the C variable age. If the firing of r1 is successful, (i.e, the relation OldPerson has been modified) the program prints a message.

4.2. The Kernel Language

4.2.1. The syntax

The rule part of an RDL/C program is composed of a set of if-then rules. The IF part of a rule is a tuple relational calculus expression. The syntax of this part is very close to the syntax of a WHERE clause in the SQL language. The THEN part of a rule is a set of *actions* that are either insertions, deletions or updates of tuples in relations. Arguments in this action part are very close to the SELECT clause of an SQL statement.

Examples :

Let Person and Worker, be two relations having the same schema (id : integer, name : char, age : integer)

The following expressions are valid LHS of rules :

```
Person(x)                                  Person(x) and Worker(y)
Person(x)  (x.age > 20)                    Exist x in Person
Person(x)  (Foreach y in Person (y.age > 20))
Person(x)  and Worker(y)  ( x.id = y.id)
Foreach x in Person (x.age  + 1 > 20)
```

The following expressions are *not* valid LHS of rules :

```
Person(x)  and Worker(x)
Person(x)  ( Exist x in Person)
Person(x)  and Worker(y)  ( x = y)
Person(x)  ( Foreach x in Person x.age > 20)
```

The Right-Hand Side (RHS) of a production supports two elementary actions, denoted "+" and "-". The update action "+" takes a set of facts and maps a database state into another state which contains these facts. On the contrary, the action "-" takes a set of facts and deletes it from a relation. A *multiple action* consists in a sequence of actions.

Examples :

The following expressions are valid RHS of rules :

```
+Person(x)                                 -Person(x)
+Person(x)  + Person(x)                    +Person(x)  - Person(x)
+Person(id =  4)                           +Person(id = x.number)
```

```
+Person(id = x.number, name = y.name, age = 20)
+Person(x) + Person(y) - Worker(x)
+Person(id = 4, name = 'Smith', age = 20)
-Person(id = 4,  age = 20, name = 'Smith')
```

The following expressions are *not* valid RHS of rules :

```
+ Person(2)                          - Person(x,2)
+ Person(x, name = 'Smith')          - Person(4, 'Smith', 20)
```

Following is a set of valid rules :

Examples :

```
If Person(x) then + Worker(x) ;                  If Person(x) then - Worker(x) ;
If Person(x) ( x.id = 4) then + Worker(x) ;  If Person(x) then + Person(x) ;
If Person(x) then + Person(x)  -Person(id= x.id, name= x.name, age= x.age +1);
If 1 = 1 then + Worker( id = 4, name = 'Smith', age = 20)
```

4.2.2. The semantics

The semantics adopted for the RDL/C language is a set oriented one: When a formula is evaluated against the database, it returns the set of instances which make the formula valid. When an action is executed against the database, it is executed for all the values which appear as arguments in the action. In the following, we present examples of rule execution.

Let us consider the following rules :

The firing of this rule leads to insert into the relation Worker the contents of the relation Person.

```
if Person(x) then + Worker(x) ;
```

The firing of this rule leads to delete from the relation R 1, the contents of the relation Person.

```
if Person(x)  then - R1(x) ;
```

The firing of this rule leads to a null action.

```
if Person(x)  then - R1(x)  + R1(x) ;
```

The firing of this rule

```
if Person(x) and R1(y) and x.att1 = 10 and x.att2 = y.att2
then +Q1(att1 = x.att1, att2 = y.att1) - R1(y) ;
```

leads to insert into the relation Q1 the following set of tuples :

```
I = {x.att1, y.att2/Person(x) and R1(y) and x.att1 = 10 and x.att2 = y.att2 }
```

and to delete from R1 the set

```
D= {x/Person(x) and R1(y) and x.att1 = 10 and x.att2 = y.att2}
```

A rule is *firable* if its condition part evaluates to True in the current database state for a particular instantiation of the free variables in the condition, and if firing it modifies the current state. *Firing*

a rule consists in modifying the current database state using the action part of the rule. More precisely, the firing operation is done as follows. First, a relational query that corresponds to the condition part of the rule is run. This query returns a result relation whose schema is given by the set of free variables in the condition part. Thus, a rule is firable whenever the query returns a non-empty result. Second, a temporary relation is built for each action encountered in the action part of the rule. It is obtained by a projection over the query result on the arguments of the action. This temporary relation is either added to or deleted from the current instance of the relation appearing in the rule.

The semantics of a rule program is then described by the following procedure :

While a firable rule exist, do fire a rule ;

The execution of a program describes a state transition diagram over database instances. A database instance is reached from a current one by firing a rule chosen at random among the set of firable rules in the current instance. Program execution terminates when no rule is firable.

4.3. Programming Constructs

A RDL/C program is composed of a set of rules as defined in the previous section. Programming constructs are added to this kernel language. These programming constructs structure programs into modules, control the execution ordering among rules, allow C main memory variables and procedural side-effects. This section presents each of these programming constructs.

4.3.1. Structuring an RDL program

The rule module is the compilation unit of the language. Its input and output interfaces are a set of base or deduced relations. The notion of module is very similar to the notion of rule set in rule-based systems. With rule sets, modular knowledge bases can be designed and the possible connection between different rule programs can be defined. Input relations are declared using the INPUT <RelationName>, ..., statement. Output relations are declared with a similar statement. The output relations are the result of a rule program execution and can be used by another rule program. Base and deduced relations are declared in the same way.

4.3.2. Partial ordering of rules

Knowledge bases need to be structured to be usable. Furthermore, the mixing of declarative reasoning and imperative control has been shown necessary for many applications. The RDL/C language allows two kinds of procedural aspects: The first one is the possibility of exchanging parameters between rule programs and C programs. The second is the possibility of specifying an explicit control over the rules. Standard control structures such as sequences and iterations can be specified with the formalism we have chosen. This formalism clarifies the control which would be otherwise spread into the rules.

The control sub-language specifies an application mode over the rules. It induces a partial ordering over the rules. Each expression of the sub-language is declared in the module. The basic terms of this language are rule names. A general expression exp in this language is :

block (exp) means that exp has to be fired until a fixpoint is reached.
seq (exp1,exp2) means that exp1 is fired once and then exp2 is fired once.

If a rule does not belong to the control language, its firing is chosen at random by the inference engine. If there is no CONTROL section, the rule interpreter applies a default strategy. If r1,r2,.., rn are the rules declared in a module, the default strategy is given by : block {r1, r2, ... ,rn} mixed with a possible partial ordering due to the stratification.

Examples

Let us consider a rule program {r1, r2, r3, r4, r5}.

seq (r1, block (r2, r3, r4), r5) is a possible expression. It enforces a computation of the form : $(r1)^a ((r2)^*(r3)^*(r4)^*)^\sigma(r5)^a$ using standard notation for context-free grammars. The notation $()^\sigma$ stands for "fire up to saturation" and a is equal to 1 or 0.

Let us consider a rule program {r1, r2, r3}.

block (seq (r1 r2 r3))enforces a computation of the form : $((r1)^a(r2)^a(r3)^a)^\sigma$ ◊

4.3.3. Main memory variables

C variables can be declared in a module. These variables are local to the module. C variables ranging over database domains can appear in a rule and are materialized as constants before firing a rule. The RDL/C inference engine is aware of these main memory variables and includes them in the semantics of rule firing. This means that the behaviour of these variables is similar to the behaviour of relations. Updates to variables referenced in a rule make the rule a candidate for firing. Variables are often used to pass parameters between rules or to communicate with the user environment. Variables used in the rule section can be simulated by a relation with a single attribute. This relation would always contain one tuple since variables are mono-valued.

Example :

Let us consider the following rule module :

```
MODULE M1 ;
BASE
            Person (name char, age integer);
DEDUCED
            OldPerson like Person;
VAR
            integer age;
RULES
r1 is
      IF Person(x) (x.age >= age)
            THEN + OldPerson(x) ;
```

```
INIT
            {scanf("%d", age);}
END MODULE
```

A C variable named age is declared in this module. It is used in rule r1 as a relation containing one tuple which is the value of the variable. This variable is initialized in the init section which is computed before the execution of the rule part of the module

4.3.4. Basic Structure of a Module

The general syntax of an RDL/C module is the following one :

```
MODULE <Module_name> ;
[INPUT <RelationName>,...;]
[BASE <RelationName>,...;]
[DEDUCED <RelationName>,...;]
[OUTPUT <RelationName>,...;]
[VAR ...;]
[REPORT
            <report specs> ]
RULES
<RuleName1> is
            {C code}
            IF <Conditions> THEN <Actions> ;
            {C code}
<RuleNameN> is
            {C code}
            IF <Conditions> THEN <Actions> ;
            {C code}
[CONTROL ...]
[INIT
            {C code} ]
[WRAPUP
            {C code} ]
END MODULE
```

After the relation declaration sections, C variables can be declared in the optional VAR section. These variables are local to the module. Their use is detailed in the following section. An optional REPORT section is used to display the results of program execution. A library of C procedures can be called in this section. These procedures take for parameters all attributes in the schema of a relation or a format parameter to display the contents of a relation. The REPORT section is executed at the end of the program. This section is followed by the RULE section.

The RULE section is a sequence of rules. Each rule is preceded by a "<RuleName> is " statement. A rule in RDL/C is preceded by an optional C statement. A C statement can be a compound C statement and hence include more that one statement. This statement is executed just before firing the rule. Then, the rule is declared and is optionally followed by C statements. This C code is executed if firing the rule has modified the database or main memory variables.

An optional control section can be declared. As described in a previous section, it consists of the string that describes an explicit ordering among rules. After the control section, Initialization statements can be used. These statements are written in C and are executed at the beginning of program execution. Wrap-up statements can also be given. These statements are written in C and are executed at the end of program execution.

4.3.5. Interfacing RDL with C

As described in the previous section, an RDL/C program includes two different languages: A production rule language where conditions and actions range over database relations and the C language which manipulates main memory variables. In this section, we describe the communication between these two languages. We also detail the effect of embedded procedural statements on the semantics of the language.

C programs to RDL/C programs:

The RDL/C compiler produces either procedures or programs. Procedures can be linked to any C program. Programs generated by the compiler are immediately executable over the database. For procedures generated by the compiler, the procedure parameters are given by the INPUT section declared in the module. The procedure result is given by the OUPTUT section also declared in the module.

RDL/C programs to C programs:

As described in the previous section, an RDL/C program can include C statements. C variables declared in the VAR section are local to the module. C variables ranging over database domains can appear in a rule and are materialized as constants before firing a rule. The RDL/C inference engine is aware of these main memory variables and includes them in the semantics of rule firing. Updates to variables referenced in a rule make the rule a candidate for firing. Variables are often used to pass parameters between rules or to communicate with the user environment. Variables used in the rule section can be simulated by a relation with a single attribute. This relation would always contain one tuple since variables are mono-valued. Special C statements might be used in a rule. The C code preceding the definition of the rule is executed just before firing the rule. The C code appearing after the definition of the rule is executed if firing the rule has been successful (i.e. it has lead to modify the database state) .

The following figures outline the possible communication between the RDL language and the C language :

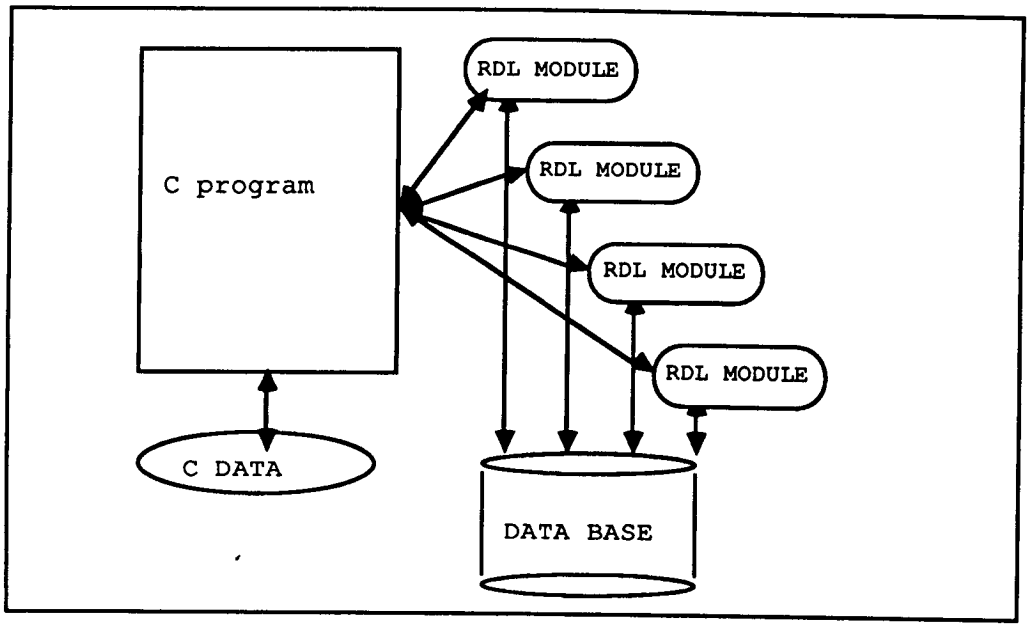


Figure 4.3 : Communication between C and RDL

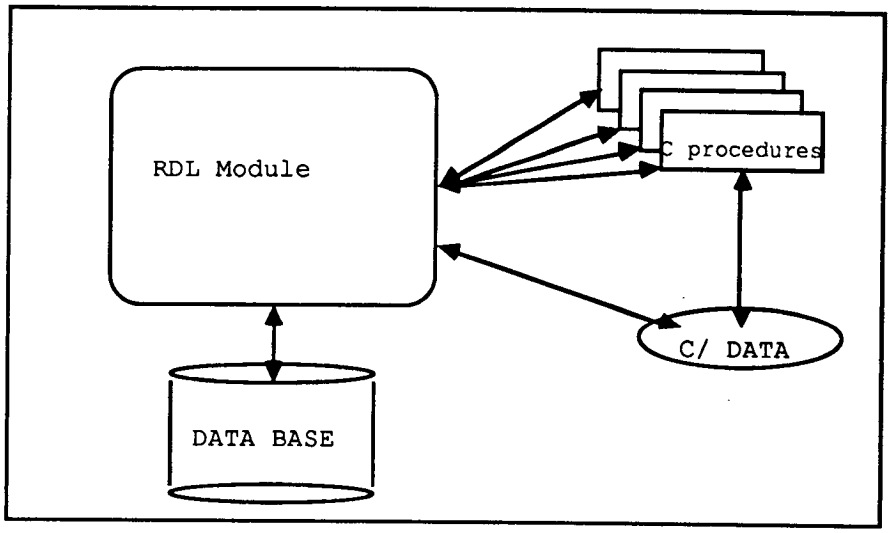


Figure 4.4 : Communication between RDL and C

5. The Language

5.1. Introduction

This section outlines each aspect of a rule module. It is intended for the RDL/C programmer. The compiler is case sensitive. However, key words may appear either in all capital letters or all in lower case letters. The list of key words are given in the following section. A word cannot have a double meaning in RDL/C. That is, no word can be used in two contexts.

5.2. Basic Elements of the language

5.2.1. Key Words

if	IF	then	THEN	thenonce	THENONCE
module	MODULE	deduced	DEDUCED	base	BASE
input	INPUT	output	OUTPUT	end	END
like	LIKE	and	AND	or	OR
not	NOT	mod	MOD	div	DIV
user	USER	use	USE	between	BETWEEN
is	IS	null	NULL	like	LIKE
escape	ESCAPE	exists	EXISTS	foreach	FOREACH
in	IN	true	TRUE	integer	INTEGER
entier	ENTIER	real	REAL	reel	REEL
char	CHAR	texte	TEXTE	control	CONTROL
block	BLOCK	seq	SEQ	report	REPORT
on	ON	header	HEADER	format	FORMAT
title	TITLE	linesppage	LINESPPAGE	rules	RULES
var	VAR	init	INIT	wrapup	WRAPUP
include	INCLUDE	extern	EXTERN		

5.2.2. Key Symbols

=	<	<=	<>	>	>=
>>	+	++	-	*	/
(	)	[	]	.	:
;	,				

5.2.3. Identifiers

Valid identifiers are symbols which are not key words and are at least two characters in length. An identifier must start with a letter and can include letters, digits, underscores or dashes. The following are valid identifiers:

r1	rep1	myRule	Start_Module	Finish_Module_	Break_22-23

5.2.4. Separators

Separators are used to isolate terms in a module. Valid separators are white spaces, tabulations, and new lines.

5.2.5. Domains

Each attribute in the schema of a relation takes its values from a particular domain. In this sense, the domain of an attribute is the type of values that can appear in that column. Standard relational systems support a limited set of domains. These include Integers, Reals and Text Strings. In the relational model that we are using, the notion of domain has been extended to include Abstract Data Types (ADT). ADT are user-defined domains. The user implements the domains which qualify his particular application. In the example we are using (the Geographic Database), the Polygon domain is an ADT. The polygon domain is used to model the geometric quality of geographic data.

Valid domains are the key words

integer	real	char
INTEGER	REAL	CHAR
entier	reel	texte
ENTIER	REEL	TEXTE

ADT domains can also appear in the body of RDL/C modules. They are a path starting at a root domain to a terminal domain name. The inheritance mechanism uses these paths when selecting functions. Root domains are the basic domain types: integer, real, and char. These are represented respectively by the sets of letters {eE}, {rR} and {tT}. A sequence of identifiers, each separated by a colon determine the remainder of the path. The following definitions are valid domains:

t:polygon	t:polygon:rectangle	t:polygon:square
e:positiveNumber	e:positiveNumber:oneToTen	r:positiveNumber

Each ADT domain appearing in a module should be known to the DBMS. For example, the t:polygon domain defines the generic type for polygons.

5.2.6. Terms

5.2.6.1. Constants

Integers

Integers are a sequence of digits (0 to 9), optionally preceded by a minus (-) sign. Valid integers are:

```
1    12    123   -45
```

Real Numbers

Real numbers are a sequence of digits (0 to 9), optionally preceded by a minus (-) sign, followed by a dot (.) and followed by a sequence of digits (0 to 9). Valid real numbers are:

```
1.0   1.1   1.11  22.999999  -24.0
```

Text Strings

Text strings are a sequence of characters delimited at each extremity by a quote (') character. Text strings cannot continue on more than one line. Valid text strings are the following:

```
'My String'      '2345'      'This is a string'
```

C Text Strings

C text strings appear in a module mostly in C statements but not in the body of pure RDL/C rules. A C string is contained between double quotes. The following are valid C strings:

```
"aString"  "a long string"
```

Type Casting

It is possible to force the type of a constant to a different type than the one that is assumed by the analyzer. This is particularly useful in the case of ADT. The casting follows the constant. The domain to which the constant is to be set if contained in square brackets. However, the base type of the constant must be compatible with the root of the ADT. That is, a text constant can be casted to an ADT whose root is 't'; an integer constant can be casted to an ADT whose root is 'e' and a real constant can be casted to an ADT whose root is 'r'. The following are valid examples of type casted constants.

```
4 [e:positiveNumber]   '(3 4)' [t:point]      4.0 [r:positiveNumber]
```

5.2.6.2. Attributes

Attributes refer to the names of columns of relations. Relations are denoted by range variables in rules. A range variable is a letter of the alphabet. Each range variable is initialized to a particular relation in the left hand side (LHS or IF part) of a rule. For example, the LHS IF EMPLOYEE(x) defines the range variable x over the EMPLOYEE relation. Attributes can then appear in the body of a rule, preceded by the range variable indicating to which relation the attribute is referring. The type of an attribute is taken from the schema of the relation in which it appears. The following are valid attribute expressions:

```
x.NAME      x.SALARY
```

5.2.6.3. Functions

Functions are Abstract Data Type (ADT) operators which can appear in the body of rules. See section 3 for more informations about ADT facilities. A function F takes one or many parameters which appear between parenthesis after the function name. Function parameters (P1, ..., Pn) are any valid expression. Parameters can thus include constants, attributes or other function calls.

```
F (P1, ..., Pn)
      where
            F is a function name, and
            each Pi is either
                  - a constant,
                  - an attribute,
                  - a function
```

```
and is of type Ti
```

Function selection considers all parameters (multi-targetting):

```
F (Ti, ..., Tn)
    where
        each Ti is a path name in the hierarchy.
```

Since function selection is done at run-time by the DBMS system, it is impossible to determine at compile time, the domain type of the function result. Hence, functions are considered as type-less by the compiler. So certain type errors will not be detected by the compiler and will only be detected at run-time. However, run-time type-error detection code is produced by the compiler to detect such type errors at run time. The following are examples of valid function calls.

```
surface ('(3 4) (5 8) (6 7) (3 4)' [t:polygon])
union (x.set, y.set)
union (x.set, intersection (x.set, y.set))
```

5.2.6.4. Variables

Variables can be declared in the variable section of a rule program and can appear in the body of rules. Variables are typed by domains which are either simple domains or ADT. Variables are either single letters or identifiers. Essentially, all variables are scalars and can assume only one value at a time. Valid variable names are the following:

```
i     MyVar     id21
```

5.2.7. Relations

The relations involved in a rule program are referenced by:

```
identifier.identifier
```

The first identifier is the name of a database and the second one is the name of the relation. The database name is optional. If it is not given, the login name is assumed for the database name. The following are valid relation names:

```
SYS.EMPLOYEE     SYS.SUPPLIERS     PARIS.INHABITANTS     EMPLOYEES
```

5.2.8. C Code

C code is C programming language source code which appears in a module. C code is one line of C source code which starts with the character ({) includes any character except a new line and ends with the character (}). Although C code cannot cross over new lines, this limitation has been overcome by allowing sequences of C code over several lines. Two valid units of C code follow:

```
{i++;}
{i++; if (i > 25) {proc1(); proc2();}}
```

5.2.9. Comments

Comments can appear anywhere in a module after the sequence (>>) up to the end of a line. Valid comments are the following:

```
>> This is a comment
r1 is >> This is another comment
```

5.2.10. Expressions

Expressions are built from terms. Terms have been described in the previous section and are constants, attributes and functions. Complex expressions are essentially built from numeric terms. Expressions are combined with the basic arithmetic operators +, -, *, /, MOD, and DIV. Parenthesis can be used to alter the priority of an operator. The type of an expression is determined from its subexpressions. Integers yield integers with the exception of the / operator. Integers and reals, or reals yield real results. Expressions of type text cannot be used with the basic arithmetic operators. Types of expressions with ADT are calculated using the ADT's root type. Types of expressions with functions become unknown and are labelled NO_TYPE and cannot be checked at compile time. The following are valid expressions:

```
3              3.0            'a string'
3 + 3          3 - 4          3 + func (4)
x.card         x.card DIV 5   x.card + 3 - func (x.a2)
```

5.3. The Production Rule

The production rule is the basic semantic unit of a module. An RDL/C program is composed of a set of if-then rules. The IF part of a rule (also called condition part) is a tuple relational calculus expression which may include main memory variables. The THEN part of a rule (also called action part) is a set of *actions* that are either insertions or deletions in a relation, variable assignments or procedural side-effects. A range restricted condition imposes that all the range variables that appear in the action part also appear positively in the condition part of the rule.

The semantics adopted for the language mixes non-deterministic and deterministic aspects. Non-determinism is in the arbitrary choice of a rule to fire at each step of program execution. Determinism is in set-oriented rules as opposed to instance-oriented rules. A rule is *firable* if its condition part evaluates to True in the current database state for a particular instantiation of the free variables of the condition, and if firing it modifies the current state. *Firing* a rule consists in modifying the current database state using the action part of the rule. More precisely, the firing operation is done as follows. First, a relational query that corresponds to the condition part of the rule is run. This query returns a result relation whose schema is given by the set of free variables of the condition part. Thus, a rule is firable whenever the query returns a non-empty result. Second, with each action (an insert, or a deletion) is built a relation, obtained by a projection over the query result on the arguments of the action. Theses relations are added or deleted to the relations figuring in the actions.

The execution of a program describes a state transition diagram over database instances. A database instance is reached from a current one by firing a rule among the set of firable rules in the current instance. A program execution terminates when no more rules are firable.

5.3.1. The IF Part

The IF or condition part of a rule contains two parts. The first part declares the range variables which will appear in the condition part of the rule. The second part is the condition itself. The condition is a tuple relational calculus expression which qualifies the tuples which will participate in the rule's firing.

5.3.1.1. Declaring Range Variables

At least one range variable must be declared in the if part of a rule. However, a range variable can appear in a quantified formula (FOREACH or EXISTS). Range variables are variables which designate virtual relations over the database. For example, the declaration:

```
SYS.EMPLOYEE (x)
```

will create one range variable called x over relation SYS.EMPLOYEE. The following declaration will create two variables called x and y over the relation SYS.EMPLOYEE:

```
SYS.EMPLOYEE (x) and SYS.EMPLOYEE (y)
```

Each variable corresponds to a virtual image of the SYS.EMPLOYEE relation.

5.3.1.2. The Condition

The condition in a rule specifies a logical condition that the tuples must satisfy in order to fire the rule. The syntax of a condition in the RDL/C language is very close to the WHERE part of a SQL statement. A rule condition is composed of a set of logical conditions connected with the logical comparators AND, OR, NOT.

The Simple Condition

Comparing Scalars

This specifies a condition between two scalars. This condition returns TRUE, FALSE or UNDEFINED.

Let us consider x as a range variable declared over the relation SYS.EMPLOYEE.

```
x.ID = 10
```

is a simple comparison between the attribute ID and the numeric constant 10. Other valid comparisons are :

```
x.NAME = 'TOM'
x.ID + 10.0 < 100
```

```
2 = 2
var1 + 1 = var2
```

where var1 and var2 are numeric C variables

```
x.ID = NB
```

where NB is a C integer

```
x.NAME < 'TOM'
length(x.GEOM) < 10.0
```

where length is a user defined function over the GEOM domain and returning a real or an integer.

More generally, a condition between two scalars follows the syntax :

```
expression1 RELOP expression2
```

where RELOP is a logical comparator chosen from the set {=, <>, <, >, <=, >=}. Expression1 and expression2 have to belong to the same type or must have compatible types such as for instance integer and real. ADT are evaluated to their basic types.

Testing for Intervals

This condition allows to test if a scalar belongs to an interval.

```
x.ID BETWEEN 100 AND 500
```

tests if x.ID belongs to the interval [100, 500].

More generally, the syntax is :

```
expression1 BETWEEN expression2 AND expression3
```

Testing for Nulls

This condition allows to test if a scalar has the NULL value, (unknown value).

```
x.ID IS NULL
```

tests if x.ID is a NULL value.

```
x.ID IS NOT NULL
```

tests if x.ID is not a NULL value.

The general syntax is :

```
expression IS [NOT] NULL
```

Text processing

It is possible to compare a textual scalar value to a pattern. For instance,

```
x.NAME LIKE '%ET'
```

returns TRUE if x.NAME is a string which terminates by the substring 'ET'.

```
x.NAME LIKE'%ET%'
```

returns TRUE if x.NAME is a string which includes the substring 'ET'.

```
x.NAME LIKE'-ET-'
```

returns TRUE if x.NAME is a string which has exactly four characters and the second character is 'E' and the third character is 'T'.

```
x.NAME LIKE'\[2]-ET-\[3]'
```

returns TRUE if x.NAME is a string which has exactly seven characters and whose third character is 'E' and whose fourth character is 'T'.

```
x.NAME LIKE'\[*]E\[*]T'
```

returns TRUE if x.NAME is a string which has 0 or n 'E' at the beginning and then 0 or n 'T' at the end of the string.

```
x.NAME LIKE'\[*]E\[*]z[' ESCAPE z
```

returns TRUE if x.NAME is a string which has 0 or n 'E' at the beginning and then 0 or n '[' at the end of the string. The escape character is used to qualify a control character as its character value.

The general syntax is :

```
expression [NOT] LIKE 'pattern' [ESCAPE 'character']
```

Quantified Formulae

Quantified formulae are allowed in the condition part of a rule. The two usual quantifiers EXISTS and FOREACH are used.

For instance :

```
EXISTS x IN SYS.EMPLOYEE
```

returns true if the relation is not empty

```
EXISTS x IN SYS.EMPLOYEE(x.ID + 1 = 200)
```

returns true if there exists a tuple x in SYS.EMPLOYEE whose attribute ID is equal to 199.

```
EXISTS x IN SYS.EMPLOYEE1, EXISTS y IN SYS.EMPLOYEE2  (x.ID = y.ID)
```

returns true if there exist a tuple x in SYS.EMPLOYEE1, a tuple y in SYS.EMPLOYEE2 such that x.ID = y.ID.

```
FOREACH x IN SYS.EMPLOYEE (x.ID <> 1000)
```

returns true if all the tuples of the relation have their ID attribute different from the value 1000.

Free variables may occur in quantified formulae:

```
SYS.EMPLOYEE(x) (EXISTS y IN SYS.EMPLOYEE (x.ID = y.ID))
```

is a valid LHS of the RDL/C language.

The general syntax of a quantified formula follows.

```
(Q1 x1 IN R1, ..., Qn IN Rn ( condition))
```

where Qi is the EXISTS or FOREACH quantifier, and each Ri is a relation.

Predicate Expressions

The Predicate Expression is a special form of Abstract Data Type. It is a function call that can only appear in the condition of a rule and whose result is a boolean value. Predicate functions appear after the key word PRED. For example, the INCLUDES predicate tests for the inclusion of a point in a polygon.

```
PRED INCLUDES ('(10 10)' [T:POLY], x.GEOM)
```

5.3.1.3. Examples of Conditions

In the following, we give examples of valid LHS of RDL/C rules. A condition immediately follows the declaration part of a rule.

- `SYS.EMPLOYEE(x)`
- `EXISTS x IN SYS.EMPLOYEE`
- `(NOT EXISTS x IN SYS.EMPLOYEE)`
- `SYS.EMPLOYEE(x) (x.ID = 100 or x.SAL > 1000)`
- `SYS.EMPLOYEE(x) ((x.SAL > 1000 and x.SAL < 2000) and (foreach y in SYS.DEPT (y.ID <> x.ID)))`
- `SYS.EMPLOYEE (x) and SYS.EMPLOYEE (y) (x.ID > y.ID)`

5.3.2. The THEN Part

The THEN part of a rule specifies the actions to perform if the rule is fired.

5.3.2.1. The Action

Insertions

An insertion adds a set of tuples in a relation. Duplicates are deleted. The argument of an insertion is an expression similar to the SQL projection clause. The name of the attribute has to be given in the argument. The ordering is not relevant.

Add to the contents of a relation

Let us consider a relation P1 having for schema: P1 (A1 integer, A2 integer, A3 char)

```
+ P1(x)
```

inserts into the relation P1 the tuples specified by the x variable ranging over a relation having the same schema as P1.

```
+ P1(A1 = 1, A2 = 2, A3 = 'A')
```

inserts the tuple (1, 2, 'A') in relation P1.

As the ordering is not relevant, this insertion is equivalent to

```
+ P1(A2 = 2, A1 = 1, A3 = 'A')
```

Constants and variables may appear in an action (var1 is a C integer):

```
+ P1(A1 = x.ID, A2 = 2, A3 = x.NAME)
+ P1(A1 = x.ID, A2 = var1, A3 = y.NAME)
```

Numeric expressions and functions are allowed in the action part:

```
+ P1 (A1 = f(x.ID) + 2, A2 = (x.ID + 4) * 4,  A3 = y.NAME)
```

Partial schema are allowed. The missing attributes are considered as NULL values:

```
+ P1 (A1 = 1 , A3 = 'A')
```

is a valid insertion and inserts the tuple (1, NULL, 'A') in the relation P1. (Note: Partial schemas are not implemented in this version; all attributes must be given)

The general syntax of an insertion is :

```
+ R1 (x)
```

where x is a range variable over a relation Q1 having same schema as R1, or

```
+ R1 (A1 = t1, ..., An = tn)
```

where A1, ..., An are attribute names of R1 and t1,..., tn are terms of compatible types.

Destroying the contents of a relation

A special insertion consists in deleting the previous contents of the relation before adding the new tuples. The syntax of this action is similar to the simple insertion except that the relation name has to be preceded by the symbol ++.

```
++ P1 (x)
```

destroys the contents of P1 before adding the set of tuples x.

The rule

```
if P1(x) and Q1(y) then - P1(x) + P1(y) ;
```

is equivalent to the rule :

```
if Q1(y) then ++ P1(y) ;
```

Note that the second rule is more efficient than the first one.

Deletions

A deletion removes a set of tuples from a relation. The syntax of a deletion is similar to an insertion except that the - symbol replaces the + symbol.

```
- P1 (A1 = 1, A2 = 2, A3 ='A')
```

deletes the tuple (1, 2, 'A') from the contents of P.

```
- P1(A1 = x.ID, A2 = var1, A3 = y.NAME)
```

deletes the set of tuples {(x.ID, var1, y.NAME)} from the contents of P.

Variable Assignments

C variable assignments are allowed in the action part of a rule.

```
var1 = 1
```

assigns the value 1 to the C variable var1

```
var1 = 1+ var2
```

assigns the expression 1+ var2 to the variable var1

```
var1 = x.ID
```

assigns the value of x.ID to the variable var1. Note that in this case, the x variable must range over exactly one tuple in a relation, if not, a run time error will occur.

The general syntax of a variable assignment is :

```
varname = expression
```

External Procedure Calls

Calls to external procedures are allowed in the action part. These calls are similar to C procedure calls in a C program. The user is in charge of checking the correct typing of the parameters. The parameters are general expressions.

```
Geometric_Display(y.XMIN,y.YMIN,y.GEOM,1,0,0)
```

is a valid call to a C procedure called Geometric_Display. The parameters are terms. The procedure is executed for each instanciation of its parameters. In this example, the y variable is ranging over the TOULOUSE.CTA relation and the constants 1, 0, 0 are X11 parameters. The complete rule is :

```
if TOULOUSE.CTA(y) then Geometric_Display(y.XMIN,y.YMIN,y.GEOM,1,0,0) ;
```

Null values for external procedure calls are handled in the following way. Attributes containing numeric nulls are zero valued while text strings are blanked (text string of length zero).

5.3.2.2. Multiple Actions

Actions may be combined together in a then part of a rule. The different elementary actions are isolated by a separator. They are sequentially executed in the order given by the user. For instance:

```
if Q1(x) (x.A1 < 10) then - P1(x) + P1(A1 = x.A1, A2 = x.A2 +1, A3 = x.A3) ;
```

deletes from P1 all the tuples which satisfy the LHS of the rule and *then*, inserts into P1 the set of tuples{(x.A1, x.A2 + 1, x.A3)}.

```
if 1 = 1 then -P1(A1 = 1, A2 = 2, A3 = 'A') + P1(A1 = 1, A2 = 2, A3 = 'B') ;
```

updates the tuple (1, 2, 'A') into (1, 2, 'B')

```
if P1(x)  (x.A1 < 10)  then -P1(x) + Q1(x) ;
```

This rule deletes from the relation P1 all the tuples which satisfy the LHS and inserts into Q1 the *same* set of tuples. That is, the LHS of the rule is *evaluated only once*, before the effects of the RHS.

Insertions, deletions, variable assignments and procedure calls may be combined together like in :

```
if SYS.EMPLOYEE(y)  (y.SAL > 1000)  then + SYS.RICH (y) - SYS.EMPLOYEE(y)
Display(y.NAME,y.SAL,y.DEPT,1,0,0) done = 1 ;
```

5.3.3. The THENONCE part

The keyword THENONCE may be used in the place on the keyword THEN. It may be followed by exactly the same action part as described above. The semantics are the following : A rule

if ... THENONCE ...;

is only fired *once*, even if the database states make it firable again. The keyword THENONCE is then equivalent to add a control variable in the RHS of the rule to control the execution:

```
r1 is
if  P1(x)  (done = 0)  then + Q1(x) ;
{puts("firing of r1"); done = 1;}
```

is equivalent to (as long as the done variable is not modified elsewhere)

```
r1 is
if  P1(x)  thenonce + Q1(x) ;
{puts("firing of r1"); }
```

5.3.4. C-code preceding the rule

A C code section can appear between the name of the rule and the IF part of the rule. The C variables appearing in this section have to be declared in the main C program or must be declared in the var section of the module. This C code is executed whenever the inference engine tries to fire the rule. It is included between the two separators {}For instance :

```
r1 is
{puts("Let's try r1");}
if ... then ...;
```

is a simple way to trace the execution of a rule program.

5.3.5. C-code following the rule

A C code section can appear at the end of a rule. The C variables appearing in this section have to be declared in the main C program or must be declared in the var section of the module. This C Code is

executed when firing the rule is successful, i.e, when firing modifies the database state. It is included between the two characters {}. For instance :

```
r1 is
if  P1(x)  (done = 0)  then + Q1(x)  ;
{puts("firing of r1"); done = 1;}
```

is a way to ensure that rule r1 will be fired only zero or one time (if the contents of done is not modified elsewhere in the program). The variable done has to be declared in the module or in the C main program which calls this module.

5.3.6. Halting the Inference Engine

The on-going inference process can be stopped by the user. It might be interesting to accomplish this in certain situations; for example, when a particular solution is found. The inference process is halted by assigning NULL to a system variable which is used by the inference engine. This assignment is done in the C-code section of a rule as follows:

```
{listeReglesPert = NULL;}
```

For example, this statement can be used to halt the rule program when rule r56 fires.

```
r56 is
     if P1(x)
     then
          +success (x);
          {listeReglesPert = NULL;}
```

5.4. The Control String

The *control section* consists of a string that describes an explicit ordering among rules. Basic symbols in this string are rule names given in the rule section. Two particular symbols BLOCK(string) and SEQ(string) can be recursively used to build a control string. The string BLOCK(string1) means that string1 must be executed up to saturation without any ordering consideration of the symbols contained in string1. On the contrary, SEQ(string1) means that string1 must be evaluated using a total ordering of the symbols in string1 from left to right. Thus, a control string can either describe a sequential or a non deterministic execution, or a combination of these [Maindreville 88].

The following is a control string declaration over the rules r1, ..., r5

```
SEQ1(r1, r2, BLOCK (r3, r4), r5)
```

This control string imposes that rule r1 fires once, then rule r2 is fired once, then rules r3 and r4 are fired up to saturation, and finally, rule r5 fires. This description assumes that all rules are firable. If a rule is not pertinent (cannot be fired), it is merely skipped and execution continues on to the following rule.

All rules that are in the control string have a greater priority of firing over rules which are not in the control string. That is, the inference engine first tries to satisfy the control string. If no rule can

be fired in the control string, the inference engine chooses a rule among the remaining pertinent rules. Pertinent rules not appearing in the control string are chosen in any order by the inference engine.

5.4.1. The Sequence Structure

The sequence structure is used when rules are to be fired in order and only once. Normally, recursive rules are fired up to saturation. A recursive rule which is present in a sequence structure is fired once and the engine tries to fire the next rule in the sequence. Consider the following rules:

```
r1 is
     if P1(x) and Q1(x)  (x.a2 = y.a1)
     then +Q1(a1 = x.a1,  a2 = y.a2) ;
r2 is
     if Q1(x) and R1(x)  (x.a2 = y.a1)
     then +R1(a1 = x.a1,  a2 = y.a2) ;
```

If there are no tuples in relation R1, as soon as rule r1 is fired, then rule r2 becomes pertinent (ie. it can be fired). However, under no specific control string, the engine might fire r1 up until saturation. If the user needs to have rule r2 fired immediately after rule r1, a control structure must be include in the rule. In the absence of a control structure, the usual way of imposing an ordering is by adding a control predicate in the body of the rules. The predicate is more often called DONE. To force a sequential firing of the rules, rules r1 and r2 could be rewritten as follows:

```
r1 is
     if P1(x) and Q1(x)  (x.a2 = y.a1 and done = 0)
     then +Q1(a1 = x.a1,  a2 = y.a2) ;
     {done = 1;}
r2 is
     if Q1(x) and R1(x)  (x.a2 = y.a1 and done = 1)
     then +R1(a1 = x.a1,  a2 = y.a2) ;
     {done = 0}
```

In the above example, the predicate done is implemented as a main memory variable which is modified in each rule's post code. Rule r1 can only fire if done = 0. If rule r1 fires, it will set done to 1 and therefore will not be able to fire until rule r2 fires. Rule r2 can only fire if done = 1 (ie. after r1 has fired). If rule r2 fires, it sets done to 0 disabling it and allowing r1 to fire again.

The presence of such predicates in the body of rules is simple for such small programs but becomes cumbersome when programs become large and the order among rules is not trivial. At that point, many such predicates have to be defined and make the program hard to read and to maintain. Moreover, there is an extra overhead involved in managing such predicates because the inference engine will try to fire a rule again unsuccessfully. The presence of a control string allows to remove such predicates from the body of rules and place them in a separate structure which is efficiently managed by the inference engine. For example, the following control string enforces the firing of rules r1 and r2 in a sequence.

```
seq (r1, r2)
```

The inference engine will fire r1, and then it will try to fire r2. This is a more efficient implementation of imposed ordering of rules because the engine will not try firing a rule twice in a row as it did with the control predicates in the above example.

Sequence strings may include Block strings which may in turn include sequence strings. It is not useful for a sequence string to include another sequence string. For example,

```
seq (r1, seq (r2, r3))
```

is equivalent to

```
seq (r1, r2, r3)
```

5.4.2. The Block Structure

The block structure imposes that rules in the block argument be fired in any order up to saturation. Consider the following control string.

```
BLOCK (r1, r2, r3)
```

This control string imposes that rules r1, r2 and r3 be fired up to saturation. Theoretically, rules in a block structure should be selected in any order; however, this implementation selects rules according to a strategy. The first rule appearing in a block is selected for firing. It is fired up to saturation. When it can no longer be fired, the engine tries to fire the next rule in the block. Once it finds a rule and fires it, the engine comes back to the first rule in the block and tries to fire it again. In essence, a rule r1 appearing to the left of a rule r2 has a higher precedence for firing. Consider the above example again. The engine fires rule r1 up to saturation. Then it tries to fire rule r2. If firing rule r2 allows rule r1 to become pertinent again, the engine will resume by firing r1. When rules r1 and r2 are no longer pertinent, the engine moves to rule r3.

It is often useful to mix seq and block control structures. Consider the following control string.

```
seq (r1, r2, block  (seq  (r3, r4)), r5)
```

This control string would impose that r1 and r2 fire in order. Then, that r3 and r4 fire in order, but up to saturation and finally, that r5 fires last.

5.5. Structure of a Module

5.5.1. Include Statements

Include statements are directives to the RDL/C compiler which specify the names of files which must be present in the compilation. Include statements are optional. These files contain C code which will be referenced in the rule module. For example, the included file may contain a procedure which implements some side-effect which is referenced in the action part of a rule (discussed later in this section). These statements precede the module name statement and are of the form *include "fileName"*. FileName is the name of the file to be included by the module. The following are valid include statements.

```
include "header.h"
include "proc.h"
```

5.5.2. The Module Name

The module statement assigns a name to the module being compiled. Module names are identifiers. The following are valid module statements.

```
module anc;
module pcc2;
```

If the module is compiled into a procedure, the name of the module will be the external interface (or procedure call) by which the module will be called from within a C program. See the compiler options section for more detail. For example, the anc module is called from a C program by the following procedure call:

```
outputRelations = anc (argc, argv, inputRelations);
```

In this procedure call, the variables outputRelations and inputRelations are of type relations (see section on rdlHeader.h) and the argc, and argv arguments are those taken from the command line and passed to the procedure. Input and output relations are discussed in the section on relation declarations.

5.5.3. The Variable Section

The variable section is used to define variables whose types are domains. Variables can be of simple domains which are integers, reals or text strings. They can also be defined over ADTs. Variables can then appear in the body of rules as terms and used in the place of constants. They are monitored by the inference engine during rule program execution. If a variable appearing in a rule changes value, the inference engine will reconsider that rule for potential firing. The variable declaration section begins with the key word VAR. A list of declarations then follow. The following is an example of a variable declaration section:

```
VAR
    integer    i,j;
    real       r1;
    char       str1;
    t:polygon  p1;
```

Integer and Real variables are initialized to 0 before running the module. Char variables are assigned a buffer of 250 characters which correspond to the maximum length of a text attribute in the DBMS we are using. An overflow of a text string may cause a run-time error. ADT variables are supported by the ADT root type. In the above example, polygons are supported by text strings. The integer domain is supported by the C int type, the real domain is supported by the C double type. Variables can be assigned values in the INIT statement. (See the section on this statement for more information). Consider for example the following INIT statement:

```
{i = 5; r1 = 1.0; strcpy(str1,"a string to be used in a rule") ;}
```

This statement will initialize variable i to the value 5, variable r1 to the value 1.0 and the variable str1 to the value "a string to be used in a rule". Strings should be assigned using the strcpy procedure because the memory for str1 has been allocated by the var declaration.

Variables can be declared as external so as to be shared by different programs. The domain type is preceded by the key word EXTERNAL to qualify variables as shared. For example, variable r1 would have been declared as external if the declaration had been:

```
VAR
                integer     i,j;
        external  real        r1;
                  char        str1;
                  t:polygon   p1;
```

5.5.4. The Relation Declaration Section

This section is used to declare all relations which will participate in the rule program. Relation names are given along with their schemas. A relation name is an identifier optionally preceded by a base name. The base name is also given as an identifier. The base name and relation name are separated by a '.'. These names correspond to actual relations which are present in the database, or to relations which have been derived by other modules or will be derived from the current module. Each relation is followed by the declaration of its schema. The schema describes the attribute names and their domains. The schema of one relation can be based on the schema of another. This avoids having to repeat the entire schema of a relation if it is the same as that of another relation. In this case, the key word LIKE is used. The following correspond to valid relation declarations:

```
road_in_fire(    ID          integer,
                 DEPARTURE   integer,
                 ARRIVAL     integer,
                 XMIN        integer,
                 YMIN        integer,
                 GEOM        t:polygon);

road_in_fire2 LIKE road_in_fire;
```

Two relations are defined by the above declaration: road_in_fire and road_in_fire2. Relation road_in_fire has six attributes. The first five are of simple domains while the last is an ADT based attribute. Relation road_in_fire2 has the same schema as road_in_fire. This is possible with the LIKE key word.

There are four types of relations in the language. These are Input, Base, Deduced and Output relations. Relations belonging to a particular class are declared in the corresponding section. Each one of these sections is optional and is described in the following.

5.5.4.1. Input Relations

Input relations are derived relations which have been produced either by an SQL program or by another module. Input relations are passed to the module in a data structure called relations (see

the rdlHeader section) which references a list of relations. The relations that the module expects to receive are given in the Input Relations Section. This section begins with the key word INPUT which is followed by a list of relations. The presence of this section in the module forces the compiler to adopt the *procedure* compiler option (see the compiler options section for more detail) which means that the module will not be compiled as an independent program but will be called as a procedure by another program. The following is a valid Input declaration section:

```
input
     road_in_fire(   ID          integer,
                     DEPARTURE   integer,
                     ARRIVAL     integer,
                     XMIN        integer,
                     YMIN        integer,
                     GEOM        t:polygon);
```

This declaration implies that the road_in_fire relation will be passed by the calling procedure in the argument list.

5.5.4.2. Base Relations

This section describes all of the base relations that will be manipulated by the module. Base relations are those relations which contain stored values in the database. Base relations are often referred to as extensional relations. The following is a valid Base relations declaration:

```
base
    TOULOUSE.CTA(   ID          integer,
                    DEPARTURE   integer,
                    ARRIVAL     integer,
                    NAME        char,
                    FONCTION    integer,
                    NBVOIES     integer,
                    DIVERS      integer,
                    LARGEUR     integer,
                    ADM         integer,
                    REVET       integer,
                    POSITION    integer,
                    XMIN        integer,
                    YMIN        integer,
                    XMAX        integer,
                    YMAX        integer,
                    GEOM        t:polygon);
```

The Toulouse.CTA relation is expected to be in the database when the rule module is run. A run-time error will be generated if the relation is not present or if its schema does not correspond to the one given in this module.

5.5.4.3. Deduced Relations

Deduced relations store intermediary results which are necessary for a rule program execution. These relations are created by the rule module and destroyed when the module is exited. This section is optional. The following is an example of a valid Deduced relation section:

```
deduced

    CTAbis LIKE TOULOUSE.CTA ;
```

5.5.4.4. Output Relations

Output relations are those produced by the rule module. Contrary to deduced relations, output relations are not freed when a module terminates. The presence of output relations assumes that the rule module will be used as a procedure called by another program. Hence, the presence of output relations will force the compiler to assume the *procedure* option as does the presence of Input relations. This optional section is started with the key word OUTPUT and followed by a list of relations. The module will return as result, a list of relations contained in the *relations* data structure described in the rdlHeader section of this manuaL The following is an example of a valid output section:

```
output
     depl_aff(  ID          integer,
                NAME        char,
                NBVOIES     char,
                FONCTION    char,
                ADM         char,
                REVET       char);
```

5.5.5. The Report Section

The report section is used to display the contents of derived relations at the end of a module run. This section is optional and starts with the key word REPORT which appears after the relation declaration section. Several reports can be listed in this section. Each report has a name (labelled with a unique identifier), and is attached to one derived relation. A title, column headers and the specification of the number of lines per page are all optional. The title, if given, is preceded by the key word TITLE. The same applies for column headers and the number of lines per page with the key words HEADER and LINESPERPAGE respectively. Finally, a print format must be given. This format is either a string which is compatible with the PRINTF display format or a procedure which will accept all of the attributes in the derived relation as parameters. Consider the following example:

```
REPORT
repl ON depl_aff IS
TITLE "This is the title of the report"
HEADER "ID\tNAME\tNBVOIES\tFONCTION\tADM\tREVET"
LINESPERPAGE 25
FORMAT "%d\t%s\t%s\t%s\t%s\t%s"
```

In the above example, a report called *rep1* is defined for the derived relation *depl_aff*. The report will be used to display all of the attributes of relation *depl_aff*. A report title will be displayed. This title is given between double quotes. A report header will appear above every page. The report header is given between double quotes. It conforms to the standards for the printf C procedure. For example, the \t will cause a TAB character to be printed between column headers. The number of lines per page will be 25 lines. If no lines are given, a default number of lines applies. Finally, a display format is given. This format is used to display the individual attributes in the

relation. The display elements in the string must correspond in number, in order and in type to the attributes in the relation. Otherwise, this may cause the program to abort. For example, %d is used to display integers and corresponds to the type of the first attribute in relation depl_aff which is of type integer. Then, %s is used to display the other attributes in the relation; these are all of type string. Null values are handled in the following way, numerics are zero valued and strings are blanked. Another possibility for displaying the contents of the relation would have been the following declaration.

```
REPORT
repl ON depl_aff IS
USE MyProc;
```

In this example, all the tuples contained in the *depl_aff* relation will be passed to procedure MyProc when the module has finished executing. This procedure must have a number of parameters equal in number and in type to the attributes in relation *depl_aff*; otherwise, a run-time error will occur. It is possible to state exactly which parameters are to be passed to a procedure; instead of having the default which is all the attributes in the relation. This is done by supplying an argument list to the stated procedure. This argument list will be used in the call instead of the default. Consider the following example:

```
REPORT
repl ON depl_aff IS
USE MyProc(NAME, var1, 34);
```

In the above example, an argument list is supplied to procedure MyProc. Therefore, for all tuples in relation *depl_aff*, procedure MyProc will be called with the value of attribute NAME, the var1 variable and the constant 34. Null value indicators can be passed to the user procedure. For each attribute in the relation, the null value indicator is the name of the attribute suffixed with *IsNull*. For example, the null value indicator for attribute NAME would be called NAMEIsNull. Null value indicators are always of type int. A value of -1 indicates the presence of a null value in the corresponding attribute.

Reports can also be run during a rule module execution for trace purposes. Consider the following rule.

```
r55 is
if P1(x) ...
then + depl_aff (x);
```

It might be interesting for the user to see the contents of the depl_aff relation each time rule r55 fires and not only its final contents at the end of the program. To obtain this effect, the following post code can be added to the rule.

```
r55 is
if P1(x) ...
then + depl_aff (x);
{repl();}
```

This procedure call will cause report repl to be executed each time rule r55 fires.

5.5.6. The Production Rule Section

The production rule is the basic semantic unit in a module. A rule module must contain at least one production rule which begins after the key word RULES. An RDL/C program is composed of a set of if-then rules. The IF part of a rule (also called condition part) is a tuple relational calculus expression which may include main memory variables. The THEN part of a rule (also called action part) is a set of *actions* that are either insertions, deletions in a relation, variable assignments or procedural side-effects. A range restricted condition imposes that all the variables that appear in the action part also appear positively in the condition part of the rule. Consider the following example.

```
rules
r1 is
  if P1(x) then +Q1(x);
```

This is a valid rule section with only one rule called r1.

5.5.7. The Control String

The control string is optional and appears after the key word CONTROL. The following is a control string declaration over the rules r1, ..., r5

```
CONTROL
     SEQ1(r1, r2, BLOC (r3, r4), r5)
```

This control string imposes that rule r1 fires once, then rule r2 is fired once, then rules r3 and r4 are fired up to saturation, and finally, rule r5 is fired. This description assumes that all rules are firable. If a rule is not pertinent (cannot be fired), it is merely skipped and execution continues till the following rule.

All rules that are in the control string have a greater priority of firing over rules which are not in the control string. That is, the inference engine first tries to satisfy the control string. If no rules can be fired in the control string, the inference engine chooses a rule among the remaining pertinent rules. Pertinent rules not appearing in the control string are chosen in any order by the inference engine.

5.5.8. The Initialization Code Section

Lines of C source code which are run before the rules start to run can be given after the control string and follow the key word INIT. This is particularly useful for initializing variables and for setting window environments before a program starts to run. Consider the following code.

```
INIT
{i = 5; strcpy (str1, "This is a string");}
{openWindow();}
```

These two lines of C code will be run before the module starts the rules. The first line sets the variable i to the value 5 and the variable str1 to the value "This is a string". The second line calls the procedure openWindow.

5.5.9. The Wrapup Section

Lines of C source code which are run after the rules have finished firing (the program has reached a fix-point) can be given in the Wrapup section. The optional wrapup section appears after the Init section and begins with the key word WRAPUP. Consider the following wrapup code.

```
WRAPUP
{printf("Type a character to quit\n");}
{getchar(); closeWindow();}
```

This C code is particularly useful at the end of a program. It avoids having the display disappear once the rules have finished firing. At the end of a module run, the message issued by the C Printf statement will appear. The program will wait for the user to type in a character. This is achieved by the call to the getchar procedure. Once the character is typed, the closeWindow procedure is called and the program is exited.

5.5.10. The End Module Statement

The end module statement should be the last statement is the RDL/C source file. Hence a module is ended with the key words END MODULE.

6. A Sample Application: A Geographic Information System

6.1. A Sample Database

6.1.1. Introduction

The RDL/C compiler has been tested over a database designed in the framework of the ESPRIT Project TROPICS. The geographic database stores information on the region of Toulouse in the south of France. In brief, the geographic model is basically the relational model with special Abstract Data Types (ADT) for geometric data. There is one general ADT for geometric data which is the polygon. The graphical elements of maps are modelled with this ADT. A polygon is represented as a list of points. To plot a map on a plane, two additional values are used. These are two integers which position the offset from where the graph should be plotted. Thus, each polygon is contained in a box or rectangle. The base of this rectangle is given by its top left coordinates which are held in two attributes, namely, XMIN and YMIN. Two additional coordinates might be given. These are the bottom right coordinates of the rectangles, namely XMAX and YMAX. The attribute which holds the graph is called GEOM. This is essentially the structure of all relations which manipulate geometric data.

A set of operators which manipulate geometric data have been implemented as ADT operators. These perform the basic operations on graphs which are used in geography. The most important ones are:

- Surface: The surface operator takes a polygon as input and returns its surface in acres.
- Length: The length operator takes a polygon as input and returns its length in kilometres.
- Adjacency: This predicate takes the offsets and geometry of two polygons and determines if they are adjacent.
- Difference: The difference operator takes the offsets and geometry of two polygons and returns the difference of the first minus the second.
- Intersection: The intersection operator takes the offsets and geometry of two polygons and returns the intersection of the two.
- Union: The union operator takes the offsets and geometry of two polygons and returns the union of the two.
- Inclusion: The inclusion operator takes the offsets and geometry of two polygons and determines if the second polygon is included in the first.

6.1.2. The Database

The Toulouse database is fourteen megabytes in size. There are eleven relations in this database. The District relation stores the districts in the region of Toulouse. There are about 70 such districts. The CTA relation stores the roads in the region. Each tuple in the CTA relation is one segment of a road from one Crossroad (intersection) to another. There are over one thousand tuples in this relation. The Crossroad relation stores all the intersections in the region of Toulouse. There are about 700 such intersections. The OS relation stores the surface occupancy for the region. The tuples in this relation plot areas as forests, lakes, urban dwellings, ... Other relations associate names to codes. This is the case for relations with names starting with Rt_. For example, the Rt_fonction relation gives the different types of usages of roads. The relations which appear in examples in the remainder of the manual are described in the following. For each relation, the contents are described; the schema of the relation is then given and finally; a sample of the tuples contained in the relation is listed.

6.1.2.1. Toulouse.CTA

This relation describes the roads in the area of Toulouse. It has sixteen attributes which are given below. The ID attribute identifies a unique segment of a road from one intersection to another. The DEPARTURE and ARRIVAL attributes describe the links between one road and another. Two tuples of this relation can be linked using these attributes (as for example in a transitive closure operation). The NAME attribute gives the name of a road. Attributes 5 to 11 give codes which qualify the road segment. Attributes 12 to 16 qualify the geometry of the road as described in the introduction.

```
 1 . ID..................... : I
 2 . DEPARTURE.............. : I
 3 . ARRIVAL................ : I
 4 . NAME................... : T
 5 . FONCTION............... : I
 6 . NBVOIES................ : I
 7 . DIVERS................. : I
 8 . LARGEUR................ : I
 9 . ADM.................... : I
10 . REVET.................. : I
11 . POSITION.............. : I
12 . XMIN................... : I
13 . YMIN................... : I
14 . XMAX................... : I
15 . YMAX................... : I
16 . GEOM................... : T:POLY
```

```
1002 tuples
+---------+----------+---------+---------+---------+---------+---------+---------+
|    ID   | DEPARTURE| ARRIVAL |   NAME  | FONCTION | NBVOIES |  DIVERS |  ...
+---+-----+----------+---------+---------+---------+---------+---------+---------+
|      570|       451|      425|I        |        4|        2|        0|
|      844|       657|      650|D24      |        4|        1|        0|
|      782|       604|      605|         |        4|        0|        0|
|      623|       488|      480|         |        4|        1|        0|
|      373|       297|      298|         |        4|        1|        0|
|      238|       195|      194|D24      |        4|        1|        0|
```

6.1.2.2. Toulouse.District

This relation describes the districts in the region of Toulouse. The INSEE attribute holds a government assigned code for the district. Attributes 2, 3 and 4 also store such codes. Attribute POPU gives the population for the district. The remainder of the attributes store geometric data for the district.

```
 1 . INSEE..................... : T(5)
 2 . ARRD...................... : I
 3 . REGION.................... : I
 4 . CODE...................... : I
 5 . NAME...................... : T
 6 . POPU...................... : I
 7 . X......................... : I
 8 . Y......................... : I
 9 . XMIN...................... : I
10 . YMIN...................... : I
11 . XMAX...................... : I
12 . YMAX...................... : I
13 . GEOM...................... : T:POLY
```

63 tuples

INSEE	ARRD	REGION	CODE	NAME	POPU	X	Y
32334	1	73	4	PUJAUDRAN	660	5040	18440
32016	1	73	4	AURADE	329	4964	18414
31182	3	73	4	FENOUILLET	2928	5239	18534
31592	3	73	4	LARRA	674	5109	18604

6.1.2.3. Toulouse.Crossroad

This relation identifies the crossroads (intersections) in the area. These intersections are points where roads meet or cross each other. The ID attribute identifies individual intersections. It can be matched against the values of attributes DEPARTURE and ARRIVAL in the CTA relation to select roads which meet at a particular intersection. The TYPE attribute qualifies the intersection. Attributes X and Y indicate its position on the map.

```
 1 . ID........................ : I
 2 . TYPE...................... : I
 3 . X......................... : I
 4 . Y......................... : I
```

701 tuples

ID	TYPE	X	Y
251	98	1527	2554
502	98	2945	1362
252	98	1371	2568
1	98	1141	4476
503	98	2948	1354
253	98	6066	2507
2	98	1224	4511

6.1.2.4. Toulouse.OS

This relation distinguishes surface occupancy in the area of Toulouse. It plots areas according to qualities identified in the attribute CODE and described in the attribute TEXTE. These can be

forests, lakes and the like. The attribute ID identifies one area. The last five attributes give the geometric information of a surface occupancy.

```
  1 . ID..................... : I
  2 . CODE................... : I
  3 . TEXTE.................. : T
  4 . XMIN................... : I
  5 . YMIN................... : I
  6 . XMAX................... : I
  7 . YMAX................... : I
  8 . GEOM................... : T:POLY
```

1178 tuples

ID	CODE	TEXTE	XMIN	YMIN	XMAX
311	8	cultures, prairies	1120	3632	1140
1036	3	espace peu artificialise'	1032	1184	1068
686	17	cours d'eau, rivie`res ou fleuves d'au moins 50m sur 1km	808	2432	840
446	3	espace peu artificialise'	2400	3244	2460
496	4	ensembles industriels et commerciaux	5080	3048	5112
93	8	cultures, prairies	3388	4212	3484
780	3	espace peu artificialise'	1444	2028	1496
138	6	grands e'quipements d'infrastructure et leur empri	5340	4100	5392
51	8	cultures, prairies	2054	4368	2104
1234	8	cultures, prairies	4358	492	4448

6.1.2.5. Toulouse.RT Fonction

This relation matches codes with descriptions of roads, in terms of their function. Seven such types have been enumerated.

```
 1 . CODE................... : I
 2 . TEXTE.................. : T
```

7 tuples

CODE	TEXTE
0	inconnu
1	type autoroutier
2	grande circulation
3	liaison regionale
4	autre
5	V.F.
6	voie industrielle

6.1.2.6. Toulouse.RT Revet

This relation matches codes with descriptions of roads, in terms of their surface covering. Ten different codes are described. They qualify road segments as segments which are regularly maintained to segments which are dirt roads.

```
  1 . CODE..................... : I
  2 . TEXTE.................... : T

10 tuples
+----------+------------------------+
|   CODE   |          TEXTE         |
+----------+------------------------+
|         0|inconnu                 |
|         7|v.f. neutralisee        |
|         1|Rte regul. entret.      |
|         8|v.f. neutralisee        |
|         2|rte non revetue         |
|         9|v.f. en construction    |
|         3|rte en constr           |
|         4|chemin d'exploitatation |
|         5|sentier                 |
|         6|v.f. en service         |
+----------+------------------------+
```

6.1.2.7. Toulouse.RT_Nbvoies

This relation matches codes with descriptions of roads, in terms of the number of lanes. Roads go from one lane to over four lanes.

```
  1 . CODE..................... : I
  2 . TEXTE.................... : T

8 tuples
+----------+------------------------+
|   CODE   |          TEXTE         |
+----------+------------------------+
|         0|inconnu                 |
|         7|2 voies larges          |
|         1|1 voie                  |
|         8|plusieurs V.F. (etranger)|
|         2|2 voies                 |
|         3|3 voies                 |
|         4|4 voies                 |
|         5|plus de 4 voies         |
+----------+------------------------+
```

6.1.2.8. Toulouse.RT_ADM

This relation matches codes with descriptions of roads, in terms of administrative jurisdiction. Roads are departmental, national highways, autoroutes or other qualifications.

```
  1 . CODE..................... : I
  2 . TEXTE.................... : T

6 tuples
+----------+------------------------+
|   CODE   |          TEXTE         |
+----------+------------------------+
|         0|inconnu                 |
|         1|SNCF                    |
|         2|autre                   |
|         3|autoroad                |
|         4|nationale               |
|         5|departementale          |
+----------+------------------------+
```

6.2. A Sample Application

6.2.1. Introduction

In the following, we present a complete sample of a geographic application written in the RDL/C language. This application uses the database schema described in section 6.1. This application first displays a part of the geographic information stored in the database. Then, the user is asked to choose on the screen a district where there is a fire. Finally, the program asks for two different crossroads to join using the shortest available roads. The program computes the shortest path between these two crossroads and displays it to the user.

This program is made of two logical parts: The first one consists in displaying the stored map. The second one is the computation of the shortest path between the chosen points. Thus, this example illustrates two different features of the language : the interaction with the C language (and the X11 window manager) and a purely computational part which is the computation of the shortest path.

The program is composed of a C main program which is in charge of initializing the graphics interface and calls different RDL/C modules to display and compute the shortest path.

6.2.2. The Main program

```
#include <stdio.h>
#include <math.h>

#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>

#include "Bd_Carto.h"
#include "iml.h"
#include "rdlHeader.h"

/*--------  Declarations  ---------*/

...
Display *Affichage;
Window   Fenetre;
GC       Contexte_Graphique;
int      Ecran;
XEvent   Evenement_X11;

XSetWindowAttributes Attribut_Fenetre;
XPoint   Tableau_de_points[NB_POINT_MAX];
int      Nombre_de_points;

float cx,cy;                    /* translation between  "map<->X11" */
int   x_District,y_District;     /* coordinates for the clicked District   */
int   xmin,xmax,ymin,ymax = 0;
double xa,ya,xd,yd;
int   xx1, yy1, x2, y2 =0;
int   origx,origy;
char  Nom_District[LG_NAME_MAX],geom[LG_GEOM_MAX];
relations *lien;
...
/*-------------------------------------------------------*/
/*                                                       */
/* This procedure extracts an array from a LISP list :   */
/*   ( (xx1 yy1) (x2 y2) ..... )                         */
/* array XPoint : { short x,y};                          */
/*-------------------------------------------------------*/
```

```
void Extrait_Coordonnees_List_Lisp(Liste_de_points,
Tableau_de_points,
Nombre_de_points,
Coef_X,Coef_Y,
Origine_X,Origine_Y)


char     *Liste_de_points;
XPoint   *Tableau_de_points;
int      *Nombre_de_points;
float     Coef_X,Coef_Y;
int       Origine_X,Origine_Y;

{
      char  Coordonnes_Temporaires[TAILLE_COORD_MAX];
      short x,y;
      int   Nombre_Courant;
      int   Indice_Temp;

      Nombre_Courant = 0;
      *Nombre_de_points = 0;

      while ( *Liste_de_points != '\0')
      {
            while  (  *Liste_de_points  ==  '('  ||  *Liste_de_points  ==  ' '  )
Liste_de_points++;

            Indice_Temp = 0;

            while ( *Liste_de_points != ')' )
                  Coordonnes_Temporaires[Indice_Temp++] = *( Liste_de_points++ );

            Coordonnes_Temporaires[Indice_Temp] = '\0';        /* le format hd signifie
*/
            sscanf(Coordonnes_Temporaires,"%hd %hd",&x,&y);
            Tableau_de_points[Nombre_Courant].x   =(short)( x*Coef_X + Origine_X );
            Tableau_de_points[Nombre_Courant++].y =(short)( y*Coef_Y + Origine_Y );

            *Nombre_de_points += 1;

            while ( *Liste_de_points == ')' || *Liste_de_points == ' ' ) Liste_de_points++
;
      }
}

/*********************************************************/
/* procedure to display a polygon       */
/*********************************************************/

Geometry_Display(x_rect_min,y_rect_min,
contour,
type,
style,
epaisseur)

int   x_rect_min,y_rect_min;
char *contour;
int   type;
unsigned int   style;
int   epaisseur;

{
...
            Extrait_Coordonnees_List_Lisp(contour,
                Polygone,
                &Nb_Sommets_Polygone,
                cx,cy,origx,origy);

            XFillPolygon(Affichage,Fenetre,Contexte_Graphique,
                Polygone,
                Nb Sommets Polygone,
```

```
                Complex,
                CoordModeOrigin);

            XFlush(Affichage);
...
}

/*******************************************************/
/* Procedure to display the name of a clicked district   */
/*                                                                    */
/*                                                                    */
/*******************************************************/

Saisie_Souris_District()

{

      int Saisie_Termine;
      int xx1,yy1;

      Saisie_Termine = False;

      while ( !Saisie_Termine )
      {
            XNextEvent(Affichage,&Evenement_X11);

            switch ( Evenement_X11.type )
            {
            case ButtonPress :
                  xx1 = Evenement_X11.xbutton.x;
                  yy1 = Evenement_X11.xbutton.y;
                  x_District = (int)(xx1/cx + x_carte_min);
                  y_District = (int)((yy1 - hauteur_fenetre)/cy + y_carte_min);
                  lien = (relations*) Search_District(0,NULL,lien);
                  XSetFillStyle(Affichage,Contexte_Graphique,FillSolid);
                  XDrawString(Affichage,Fenetre,Contexte_Graphique,
                      xx1,yy1,Nom_District,strlen(Nom_District));
                  XFlush(Affichage);
                  Saisie_Termine = True;
                  break;
            case KeyPress :
                  Saisie_Termine = True;
                  break;
            default :
                  break;
            }
      }
}

/*******************************************************/
                        MAIN PROGRAM

/*******************************************************/

main(argc,argv)
int argc;
char **argv;
{
      float c_y;
      int alpha,beta;
      int Refus = -1;
      lien = NULL ;

/* Connection to the database system*/

      if ( Connexion_a_la_Base() == Refus )
      {
            fprintf(stderr,"\n\n   Connection   to   the   SABRINA   DBMS   impossible.
Exiting.....\n");
            exit(0);
      }
```

```
...

X11_Init();

/* Open the main X11 window where the results are displayed */

Ouvre_Fenetre_X11();

cx =  (float)largeur_fenetre/beta;
cy = -cx;

puts("displaying of the districts");
/* Call to the Districts module */

(void)Districts(argc, argv,NULL);

puts("displaying of the forests");
/* Call to the Verdure module */

(void)Verdure(argc,argv,NULL);

puts("displaying of the inhabited areas");
/* Call to the Urbdens module */

(void)Urbdens(argc,argv,NULL);

puts("fire area input");

Saisie_Souris_District();
xmin=xmax=x_District;
ymin=ymax=y_District;

puts("display of the roads in fire");

/* Call to the Roads_in_fire module */

lien=(relations*) Roads_in_fire(argc, argv, lien);

puts("input the starting point");
Saisie_Souris_District();
xd=xmin=xmax=x_District;
yd=ymin=ymax=y_District;

/* Call to the Departure_roads module */

lien=(relations*) Departure_roads(argc, argv, lien);

puts("input the arrival point);
Saisie_Souris_District();

/* Call to the Arrival_roads module */
lien=(relations*) Arrival_roads(argc, argv, lien);

/* Call to the Pcc2 module */
lien= (relations*) Pcc2(argc,argv,lien);

(void)getchar();

X11_Close();
}
```

Important items in this program have been flagged in bold characters. At the beginning of the program, the files iml and rdlHeader have been included. The variable lien has been declared of type relations. This type is defined in the rdlHeader file. It will be used to pass input relations to the rule programs and obtain output result relations.

In the main program section, the District module is called to display the districts in the Toulouse database. The Verdure module is called to display those areas on the map where green spaces can be found. The Urbdens module is called to display those areas which are high density urban dwellings. Then, the user is asked to select the district in which the fire is located. The module Search_District is used to locate the district where the mouse was clicked. The module Roads_in_fire is called to select those roads which are in the district where the fire is located. Roads_in_fire also darkens this district on the map. The user is asked to select the departure district and the arrival district. The roads in these respective districts are isolated by the Departure_roads and the Arrival_roads modules respectively. Finally, module Pcc2 is called to find a good path between the departure and the arrival without going through the roads in the district where the fire is located.

6.2.3. The Rule Modules

In the following, we present the different RDL/C rule modules which are used in the geographic application. These rule modules are called from the main C program described above. The first rule modules are used to select part of the geographic information which is stored in the database. The Pcc2 module is the one which computes the shortest path between two clicked districts.

6.2.3.1. Search_District

This module takes as input the geographic coordinates of a clicked point on the screen. These coordinates are stored in the x_District and y_District C variables. It returns the Nom_District variable which contains the name of the district where the mouse was clicked. It affects also the contents of four global variables xmin, xmax,ymin, ymax which are used to store the geometry of a district. The contents of these variables is modified in the RHS of the rule r3.

```
module Search_District;
var
      extern integer x_District,y_District;
      extern char Nom_District ;
base
    TOULOUSE.DISTRICT(    INSEE       char,       ARRD   integer,
                          REGION      integer,    CODE   integer,
                          NAME        char,       POPU   integer,
                          IX          integer,    IY     integer,
                          XMIN        integer,    YMIN   integer,
                          XMAX        integer,    YMAX   integer,
                          GEOM        t:poly);

deduced
    comm_inter like TOULOUSE.DISTRICT;
output
    District like TOULOUSE.DISTRICT;


rules

r1 is

   if TOULOUSE.DISTRICT(x)
      (x.XMIN < x_District and  >> inserts in the relation com_inter all the
       x.XMAX > x_District and  >> Districts that intersect x_District y_District
```

```
      x.YMIN < y_District and
      x.YMAX > y_District)

   then
      +comm_inter(x);

r2 is

   if comm_inter(x)
      (xyinpoly(x_District, y_District, geom_to_poly(x.XMIN, x.YMIN, x.GEOM))='T' )
   then

      ++District(x)
      Nom_District= x.NAME ; >> modifies the contents of the Nom_District variable

r3 is

  if District (x)
  then

      aff_minmax(x.XMIN, x.YMIN, x.XMAX, x.YMAX);
      >> call to the aff_minmax procedure to modify the xmin, xmax, ymin, ymax variables

init
{strcpy(Nom_District,"");}

end module
```

Rule r1 selects the districts which may contain the point where the mouse was clicked. This is done using attributes which give the coordinates of the smallest rectangle containing the district. These districts are assigned to the deduced relation comm_inter. Rule r2 then takes the actual geometry of each district selected by the preceding rule and matches the point where the mouse was clicked to the geometry of the district. This is achieved with the xyinpoly function. Rule r3 assigns the district area to variables.

6.2.3.2. Locate Fire

This module displays on the screen the geometry of the district with the fire and stores in the road_in_fire relation the roads which are not available (ie. the roads which are in the district with the fire).

```
module Roads_in_fire;

input
      District(        INSEE       char,       ARRD        integer,
                       REGION      integer,    CODE        integer,
                       NAME        char,       POPU        integer,
                       IX          integer,    IY          integer,
                       XMIN        integer,    YMIN        integer,
                       XMAX        integer,    YMAX        integer,
                       GEOM        t:poly);

base
  TOULOUSE.CROSSROAD(  ID          integer,    TYPE        integer,
                       XC          integer,    YC          integer);

    TOULOUSE.CTA(      ID          integer,    DEPARTURE   integer,
                       ARRIVAL     integer,    NAME        char,
                       FONCTION    integer,    NBVOIES     integer,
                       DIVERS      integer,    LARGEUR     integer,
                       ADM         integer,    REVET       integer,
```

```
                        POSITION    integer,    XMIN        integer,
                        YMIN        integer,    XMAX        integer,
                        YMAX        integer,    GEOM        t:poly);
deduced

    inter_fire  like TOULOUSE.CROSSROAD;
    inter_fire2 like TOULOUSE.CROSSROAD;

output

      road_in_fire(     ID          integer,    DEPARTURE   integer,
                        ARRIVAL     integer,    XMIN        integer,
                        YMIN        integer,    GEOM        t:poly);


rules

r1 is

   if District(x) and TOULOUSE.CROSSROAD(y)
      (x.XMIN < y.XC and
       x.XMAX > y.XC and
       x.YMIN < y.YC and
       x.YMAX > y.YC)
   then

      +inter_fire(y); >> stores in the inter_fire relation the in fire crossroads

r1bis is

  if District (y)
  then

      Geometry_Display(y.XMIN, y.YMIN, y.GEOM,1,6,0); >> displays the in fire District

r2 is

  if District(x) and inter_fire(y)
      (xyinpoly (y.XC, y.YC, geom_to_poly (x.XMIN, x.YMIN, x.GEOM)) = 'T')
  then

      + inter_fire2(y);

r3 is

  if TOULOUSE.CTA(x) and inter_fire2(y)  (x.DEPARTURE = y.ID or x.ARRIVAL = y.ID)
  then

      >> inserts in the relation road_in_fire all the in fire roads
      +road_in_fire(    ID          = x.ID,
                        DEPARTURE   = x.DEPARTURE,
                        ARRIVAL     = x.ARRIVAL,
                        XMIN        = x.XMIN,
                        YMIN        = x.YMIN,
                        GEOM        = x.GEOM);

end module
```

Rule r1 selects the intersections in the Toulouse.Crossroad relation which are in the rectangle containing the district that was selected by preceding module. These intersections are put into the inter_fire relation. Rule r1bis displays in grey, the district with the fire. Rule r2 maps each intersection selected by the previous rule to the actual geometry of the district. Rule r3 selects all roads in the CTA relation which either arrive or leave the intersections selected by the preceding rule and puts into the road_in_fire relation, all roads in the selected district.

6.2.3.3. Locate Departure

This module computes all the roads starting from the chosen district. These are the departure point for the calculation of the shortest path. The logic behind this module is similar to the previous one.

```
module Departure_roads;

input
      District (          INSEE       char,       ARRD        integer,
                          REGION      integer,    CODE        integer,
                          NAME        char,       POPU        integer,
                          IX          integer,    IY          integer,
                          XMIN        integer,    YMIN        integer,
                          XMAX        integer,    YMAX        integer,
                          GEOM        t:poly);

base
  TOULOUSE.CROSSROAD(     ID          integer,    TYPE        integer,
                          XC          integer,    YC          integer);

    TOULOUSE.CTA (        ID          integer,    DEPARTURE   integer,
                          ARRIVAL     integer,    NAME        char,
                          FONCTION    integer,    NBVOIES     integer,
                          DIVERS      integer,    LARGEUR     integer,
                          ADM         integer,    REVET       integer,
                          POSITION    integer,    XMIN        integer,
                          YMIN        integer,    XMAX        integer,
                          YMAX        integer,    GEOM        t:poly);

deduced

    inter_depart  like TOULOUSE.CROSSROAD;
    inter_depart2 like TOULOUSE.CROSSROAD;

output
      road_depart (       ID          integer,    DEPARTURE   integer,
                          ARRIVAL     integer,    XMIN        integer,
                          YMIN        integer,    GEOM        t:poly);



rules

r1 is

   if District(x) and TOULOUSE.CROSSROAD(y)
      (x.XMIN < y.XC and
       x.XMAX > y.XC and
       x.YMIN < y.YC and
       x.YMAX > y.YC)
   then

      +inter_depart(y);

r2 is

  if District(x) and inter_depart(y)
      (xyinpoly (y.XC, y.YC, geom_to_poly (x.XMIN, x.YMIN, x.GEOM)) = 'T')
  then

      >> computes the departure District
      + inter_depart2(y);

r3 is

  if TOULOUSE.CTA(x) and inter_depart2(y) (x.DEPARTURE = y.ID or x.ARRIVAL = y.ID)
```

```
 then

      >>  computes all the departure roads
      +road_depart(       ID          = x.ID,
                          DEPARTURE   = x.DEPARTURE,
                          ARRIVAL     = x.ARRIVAL,
                          XMIN        = x.XMIN,
                          YMIN        = x.YMIN,
                          GEOM        = x.GEOM);

r4 is

  if road_depart(x)
  then

      >>  displays the departure roads
      Geometry_Display(x.XMIN, x.YMIN, x.GEOM, 1, 0, 0)
      Geometry_Display(x.XMIN, x.YMIN, x.GEOM, 0, 0, 1);


end module
```

6.2.3.4. Locate Arrival

This module computes the roads which are in the district which is designated as the arrival.

```
module Arrival_roads;

input
>> COMMUNE is DISTRICT

      District(          INSEE       char,       ARRD        integer,
                         REGION      integer,    CODE        integer,
                         NAME        char,       POPU        integer,
                         IX          integer,    IY          integer,
                         XMIN        integer,    YMIN        integer,
                         XMAX        integer,    YMAX        integer,
                         GEOM        t:poly);

base
>>TOULOUSE.CARREFOUR is TOULOUSE.CROSSROAD

  TOULOUSE.CROSSROAD(    ID          integer,    TYPE        integer,
                         XC          integer,    YC          integer);

    TOULOUSE.CTA(        ID          integer,    DEPARTURE   integer,
                         ARRIVAL     integer,    NAME        char,
                         FONCTION    integer,    NBVOIES     integer,
                         DIVERS      integer,    LARGEUR     integer,
                         ADM         integer,    REVET       integer,
                         POSITION    integer,    XMIN        integer,
                         YMIN        integer,    XMAX        integer,
                         YMAX        integer,    GEOM        t:poly);

deduced

    inter_arrival  like TOULOUSE.CROSSROAD;
    inter_arrival2 like TOULOUSE.CROSSROAD;

output
      road_arrival(      ID          integer,    DEPARTURE   integer,
                         ARRIVAL     integer,    XMIN        integer,
                         YMIN        integer,    GEOM        t:poly);
```

```
rules

r1 is

   if District(x) and TOULOUSE.CROSSROAD(y)
      (x.XMIN < y.XC and
       x.XMAX > y.XC and
       x.YMIN < y.YC and
       x.YMAX > y.YC)
   then

      +inter_arrival(y);

r2 is

  if District(x) and inter_arrival(y)
      (xyinpoly (y.XC, y.YC, geom_to_poly (x.XMIN, x.YMIN, x.GEOM)) = 'T')
  then

      + inter_arrival2(y);

r3 is

  if TOULOUSE.CTA(x) and inter_arrival2(y)
     (x.DEPARTURE = y.ID or x.ARRIVAL = y.ID)
  then

      +road_arrival(     ID          = x.ID,
                         DEPARTURE   = x.DEPARTURE,
                         ARRIVAL     = x.ARRIVAL,
                         XMIN        = x.XMIN,
                         YMIN        = x.YMIN,
                         GEOM        = x.GEOM);

end module
```

6.2.3.5. Good Path

This module computes a good path between the departure and arrival districts, avoiding those roads which are in the district on fire.

```
module Pcc2;

var
  extern integer xmin, xmax, ymin, ymax;
  extern integer x_District, y_District;
  integer gen_id, done;

input
      road_in_fire(      ID          integer,    DEPARTURE   integer,
                         ARRIVAL     integer,    XMIN        integer,
                         YMIN        integer,    GEOM        t:poly);

      road_depart(       ID          integer,    DEPARTURE   integer,
                         ARRIVAL     integer,    XMIN        integer,
                         YMIN        integer,    GEOM        t:poly);

      road_arrival(      ID          integer,    DEPARTURE   integer,
                         ARRIVAL     integer,    XMIN        integer,
                         YMIN        integer,    GEOM        t:poly);
```

```
base
      TOULOUSE.CTA(      ID          integer,    DEPARTURE   integer,
                         ARRIVAL     integer,    NAME        char,
                         FONCTION    integer,    NBVOIES     integer,
                         DIVERS      integer,    LARGEUR     integer,
                         ADM         integer,    REVET       integer,
                         POSITION    integer,    XMIN        integer,
                         YMIN        integer,    XMAX        integer,
                         YMAX        integer,    GEOM        t:poly);

    toulouse.rt_fonction(CODE        integer,    TXT         char);

    toulouse.rt_revet(   CODE        integer,    TXT         char);

    toulouse.rt_adm(     CODE        integer,    TXT         char);

    toulouse.rt_nbvoies( CODE        integer,    TXT         char);

deduced

      CTAbis like TOULOUSE.CTA;

      steps (            ID          integer,    CTA         integer,
                         DEPARTURE   integer,    ARRIVAL     integer,
                         LENGTH      integer,    DISTANCE    real,
                         RAMIFIE     integer);


      newsteps(          ORIGINE     integer,    ID          integer,
                         CTA         integer,    DEPARTURE   integer,
                         ARRIVAL     integer,    DISTANCE    real,
                         LENGTH      integer);

      good_steps(        CTA         integer,    DEPARTURE   integer);

      delta (            CTA         integer,    DEPARTURE   integer);

      steps_aff(         ID          integer,    NAME        char,
                         NBVOIES     char,       FONCTION    char,
                         ADM         char,       REVET       char);


report
  rep3 on steps_aff is
  title "\t\tsteps"
  header "ID\tNAME\tNBVOIES\tFONCTION\t\tADM\t\tREVET"
  linesppage 25
  format "%d\t%-7s\t%-7s\t%-20s %-20s %-20s";


rules


init_traj is

      >>Eliminate all roads which are not in the global rectangle
      {puts("Examining init_traj");}

  if TOULOUSE.CTA(x)
    ((x.XMIN > xmin) and (x.XMAX < xmax) and (x.YMIN > ymin) and (x.YMAX < ymax)
   and (foreach y in road_in_fire (y.ID <> x.ID)))
  thenonce

      + CTAbis(          ID          = x.ID,
                         DEPARTURE   = x.DEPARTURE,
                         ARRIVAL     = x.ARRIVAL,
                         NAME        = x.NAME,
                         FONCTION    = x.FONCTION,
                         NBVOIES     = x.NBVOIES,
                         DIVERS      = x.DIVERS,
```

```
                    LARGEUR      = x.LARGEUR,
                    ADM          = x.ADM,
                    REVET        = x.REVET,
                    POSITION     = x.POSITION,
                    XMIN         = x.XMIN,
                    YMIN         = x.YMIN,
                    XMAX         = x.XMAX,
                    YMAX         = x.YMAX,
                    DISTANCE     = sqr ((x.XMIN + x.XMAX) /2 - x_District)
                         + sqr ((x.YMIN + x.YMAX) /2 - y_District),
                    GEOM         = x.GEOM);
     {puts("Firing init_traj");}


init_trajbis is

     >> The relation steps is initialized with the departure roads
     {puts("Examining init_trajbis");}

 if CTAbis(x) and road_depart (y) (x.ID=y.ID)
 thenonce

     +steps(      ID           = gen_id*1000+x.ID,
                  CTA          = x.ID,
                  DEPARTURE    = x.DEPARTURE,
                  ARRIVAL      = x.ARRIVAL,
                  LENGTH       = longueur(x.GEOM),
                  DISTANCE     = x.DISTANCE,
                  RAMIFIE      = 0) ;

     {gen_id++;}
     {puts("Firing init_trajbis");}


init_trajbisbis is

     >> The relation steps is initialized with the starting roads, in the other direction
     {puts("Examining init_trajbisbis");}

 if steps(x)
 thenonce

     +steps(      ID           = gen_id*1000+x.ID,
                  CTA          = x.ID,
                  DEPARTURE    = x.ARRIVAL,
                  ARRIVAL      = x.DEPARTURE,
                  LENGTH       = x.LENGTH,
                  DISTANCE     = x.DISTANCE,
                  RAMIFIE      = 0) ;

     {gen_id++;}
     {puts("Firing init_trajbisbis");}


ramif is

     >> computes a new step
     {puts("Examining ramif");}

 if steps(x) and CTAbis(y)
           (x.ARRIVAL = y.DEPARTURE and
           y.ID <> x.CTA and
           y.DISTANCE < x.DISTANCE and
           x.RAMIFIE = 0)
 then

     ++newsteps(  ORIGINE      = x.ID,
                  ID           = gen_id*1000 + y.ID,
                  CTA          = y.ID,
                  DEPARTURE    = y.DEPARTURE,
                  ARRIVAL      = y.ARRIVAL,
```

```
                    DISTANCE      = y.DISTANCE,
                    LENGTH        = x.LENGTH + longueur(y.GEOM)) ;

      {gen_id++;}
      {done = 1;}
      {puts("Firing ramif");}


ramifbis is

      >> compute a new step
      {puts("Examining ramifbis");}

 if steps(x) and CTAbis(y)
            (x.ARRIVAL = y.ARRIVAL and
            y.ID <> x.CTA and
            y.DISTANCE < x.DISTANCE and
            x.RAMIFIE = 0)
 then

      +newsteps(    ORIGINE       = x.ID,
                    ID            = gen_id*1000+y.ID,
                    CTA           = y.ID,
                    DEPARTURE     = y.ARRIVAL,
                    ARRIVAL       = y.DEPARTURE,
                    DISTANCE      = y.DISTANCE,
                    LENGTH        = x.LENGTH + longueur(y.GEOM) ) ;

      {gen_id++;}
      {done =1;}
      {puts("Firing ramifbis");}


aff_geom is

      >> displays the new steps

if newsteps (x) and CTAbis(y) (x.CTA = y.ID)
then

      Geometry_Display(y.XMIN, y.YMIN, y.GEOM,1,0,0)
      Geometry_Display(y.XMIN, y.YMIN, y.GEOM,0,0,1);


maj is

      >> inserts in  the relation steps the new steps
      {puts("Examining maj");}

  if newsteps(x) and steps(y) (x.ORIGINE = y.ID and done =1)
  then

      +steps(       ID            = x.ID,
                    CTA           = x.CTA,
                    DEPARTURE     = x.DEPARTURE,
                    ARRIVAL       = x.ARRIVAL,
                    LENGTH        = x.LENGTH,
                    DISTANCE      = x.DISTANCE,
                    RAMIFIE       = 0)

      -steps(y)

      +steps(       ID            = y.ID,
                    CTA           = y.CTA,
                    DEPARTURE     = y.DEPARTURE,
                    ARRIVAL       = y.ARRIVAL,
                    LENGTH        = y.LENGTH,
                    DISTANCE      = y.DISTANCE,
                    RAMIFIE       = 1);

      {puts("Firing maj");}
```

```
      (done = 0;)


optim is

      >> deletes the longest paths from the steps relation
      (puts("Examining optim");)

  if steps(x) and steps(y)
            ((x.ARRIVAL = y.ARRIVAL) and
            x.LENGTH < y.LENGTH)
  then

      -steps(y);

      (puts("Firing optim");)


end_init is

      >> This rule is fired when there exists a road between the departure and arrival
      (puts("Examining end_init");)

  if steps(x) and road_arrival(z)
      (x.CTA = z.ID)
  thenonce

      +good_steps(        CTA          = x.CTA,
                          DEPARTURE    = x.DEPARTURE)
      -steps(x);

      (puts("Firing end_init");)


end_copy is

      >> copies all the elementary steps in the good_steps relation
      >> computation of the transitive closure of the relation steps
      (puts("Examining end_copy");)

   if steps(x) and good_steps(y)
     (x.ARRIVAL = y.DEPARTURE)
   then

      +good_steps(        CTA          = x.CTA,
                          DEPARTURE    = x.DEPARTURE) ;

      (puts("Firing end_copy");)


copy_aff is

      >> displays the shortest path on the screen
      (puts("Examining copy_aff");)

   if good_steps(x) and CTAbis(y)
      (x.CTA = y.ID)
   then

      Geometry_Display(y.XMIN, y.YMIN, y.GEOM,0,0,2);

      (puts("Firing copy_aff");)


copy_affbis is

      >> displays the travel information about the roads between the two crossroads
      (puts("Examining copy_affbis");)

    if good_steps(x) and CTAbis(y) and
      toulouse.rt_fonction(a) and
```

```
      toulouse.rt_revet(b) and
      toulouse.rt_adm(c) and
      toulouse.rt_nbvoies(e)
      (x.CTA = y.ID
       and a.CODE = y.FONCTION
       and b.CODE = y.REVET
       and c.CODE = y.ADM
       and e.CODE = y.NBVOIES)
    then

      +steps_aff( ID          = y.ID,
                  NAME        = y.NAME,
                  NBVOIES     = e.TXT,
                  FONCTION    = a.TXT,
                  ADM         = c.TXT,
                  REVET       = b.TXT) ;

      {puts("Firing copy_affbis");}
      {listeReglesPert = NULL;}


control

      seq (init_traj, init_traj[illegible]_trajbisbis,
      block (seq (ramif,ramifbi[illegible]eom,maj,optim, end_init,
                  block (end_co[illegible]opy_aff, copy_affbis))) ;


init
      {printf("Starting Pcc2\n");}
      {gen_id=1; done = 0;}


end module
```

The first rule init_traj puts into CTAbis the roads that will participate in the calculation of a good path. These are the roads in a rectangle on the map calculated from the districts where the mouse was clicked. For each road selected with this method, the road must not be found in the set of roads in the district where the fire is found. The second rule init_trajbis initializes the steps relation with the first displacement; these are the roads found in the departure relation. The init_trajbisbis module takes all of the roads selected by the previous relation and adds the same roads inverting the departure and arrival attributes. This adds to the step relation reverse roads to disregard directions. The ramif and ramifbis rules calculate a newstep from each step to determine connections between roads. This is a typical transitive closure operation to establish connections between roads. The aff_geom rule is used to display the new roads that have been calculated. The maj rule adds to the step relation all of the new roads which have joined with the previous roads. The optim rule removes all paths which, for a same arrival have a longer length. The end_init rule signals the presence of a path. The end_copy rule backtracks from the arrival back to the starting point to establish the path that has been selected by the program. The copy_aff rule displays a thick line corresponding to this path. The copy_affbis rule reports the roads selected by the program (name, type, ...).

The control string insures that the rules ramif,ramifbis,aff_geom,maj and optim are done in sequence. For the program to work correctly, this must be the case. When end_init is able to fire, the

remainder of the rules in the program can fire to display and report a path from the departure to the arrival. The C code of copy_aff stops the rule program. This is done with

```
{listeReglesPert = NULL;}
```

which is the C variable which holds the list of pertinent rules.

6.2.4. Results of Execution

In this section, we give some results of the execution of the RDL/C program. The figures described in the following can be found at the end of the manual. The first figure displays the map which is stored in the database. This screen is displayed after the connection to the database without any interaction with the user. It represents the contour of the different districts, the forests, and the inhabited areas. It is displayed by the execution of the modules Districts, Verdure and Urbdens.

The second figure represents the state of the main window after the execution of the Roads_in_fire, Departure_roads and Arrival_roads modules. As can be seen on the figure, the names of the departure, arrival and on fire districts are displayed to the screen. The on fire district is shaded in grey.

The third figure represents the main window at the end of execution. The shortest path is drawn in bold.

The fourth figure is also one complete execution with the associated information on the shortest path, i.e, the name of the different roads, the type of roads, etc.

The final figure is a trace of the execution of a program. These messages are included in the RDL/C modules and display the name of the current rule to fire.

7. Compiler Options

The source code of an RDL/C module is written in a file which should have the suffix .r. The general architecture of the RDL/C compiler is portrayed in Figure 7.1.

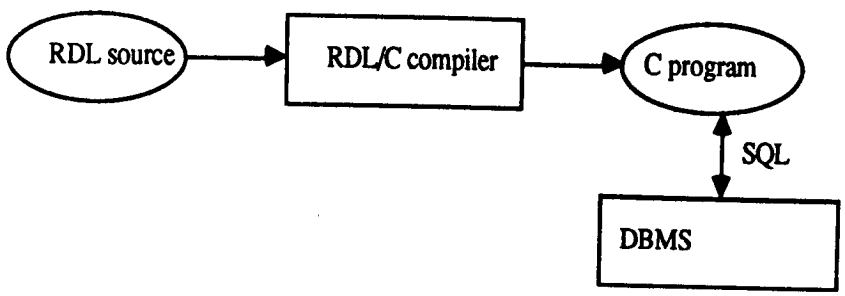


Figure 7.1. Sketch of the RDL/C compilation and run-time environment.

The compiler accepts a source program and produces as output, a C program which implements the rule program. The DBMS does not require any inferencing capabilities to process the program. The C program contains code to implement each rule and includes the inference engine which fires rules until a fixpoint is reached. All data remains in the DBMS during the inference process. This is because rules are based on relational calculus and can thus be solved by the DBMS.

There exist different options to compile a rule module. These options are detailed in the next sections. The user interface includes two sets of primitives: The SQL language, and a set of commands to edit, run, compile and debug rule programs. There are two key compilation options. The first one generates a C/SQL procedure. The input interface to these procedures is given by the INPUT section declared in the module. The output interface is given by the OUTPUT section also declared in the module. The second option generates C/SQL programs. The option -c creates the C program produced by the RDL/C compiler. The program will be created with the same name as the source except that it will have the suffix .c instead of the suffix .r. However, using the -c option inhibits the compilation of the program into its executable version. This option can be used to see how the compiler implements rule programs in C. The name of the RDL/C source file is stated on the RDL/C command line along with other object code files and libraries used in the compilation. Programs produced by the RDL/C compiler can be run with the -t option to obtain run-time information such as the list of pertinent rules for a cycle; the rule chosen by the inference engine for firing; the number of tuples produced by a rule's firing; the SQL query produced by the inference engine; and error messages. RDL/C source and object code are managed by the UNIX standard editors and file system.

7.1. The Module as a Program

A RDL/C module can be compiled in an executable application program. The syntax is :

```
rdlC name_of_file.r
```

Example : rdlC pcc1.r

The command line must contain one and only one .r file. If other files and libraries are needed in the compilation, they can also be included on this line. Consider the following example:

```
rdlC modl.r draw.o /usr/lib/libXaw.a
```

The draw.o file could have been produced by the C compiler from a C source file called draw.c. The command for compiling this file is the following one.

```
cc -c draw.c
```

Any number of these .o files can appear on the command line. Also, special libraries such as the X-Window library can be required. These files also appear on the command line.

If no errors are detected in the compilation, it will result in an executable file whose name is the name of the .r file.

7.2. The Module as a Procedure

A RDL/C module can be compiled into a procedure. The corresponding option is -pr. This option will compile a .r file into a .o file which is to be linked with a C main program. The presence of INPUT or OUTPUT relations in the module will automatically switch on this option. This is discussed in the next section on compiling modules and C programs. The execution status of a module is set in the imlsqlcode variable and can be checked after the module call. A status value of zero indicates that no errors or problems occurred during the module's execution.

Example: rdlC -pr Pcc2.r produces a compiled version of a C procedure Pcc2(argc, argv, relations) where *relations* is a list of relations (defined in the section on the rdlHeader file) corresponding to the INPUT section. This procedure returns a list of relations corresponding to the OUTPUT section. For more information, see the relation declaration section of this manual.

Example: (void)Verdure(argc,argv,NULL); is a valid call to the compiled version of the Verdure RDL/C module. The NULL argument means that no input relations are used in this module. The Verdure module should not return any relations.

Example : lien=(relations*) Pcc2(argc, argv, lien); is a valid call to the compiled version of the Pcc2 module. The execution of this procedure returns a list of relations.

7.3. Compiling Modules and C programs

If modules have been compiled as procedures, they must be linked together with a C main program. The command to compile an application from a C main program and from modules is the Crdl command. Consider the following command.

```
Crdl main.c Pcc2.o draw.o /usr/lib/libXaw.a
```

There can only be one .c file on the command line. The remaining files can be modules compiled as procedures. This is the case for file Pcc2.o. Other files can also be C program files to be included in the application. This is the case for the draw.o file. Libraries may also appear on the command line. The name of the application is the name of the .c file. In this case, the application will be called main.

The main program must include the DBMS connect and disconnect commands. The body of the main program should look like the following.

```
#include "iml.h"
#include "rdlHeader.h"
char user[100], passwd[100];
relations *liens;
...
/* start program */
strcpy (user, "SYS");
strcpy(passwd, "");

rep = sql_connection (user, passwd);
if (rep != 0) exit (-1);
/* body of program */
...
liens = Pcc2(argc, argv, liens);
...
/* end of program */
rep = sql_deconnection();
exit;
```

The iml.h file includes all the declarations necessary to communicate with the DBMS. (See the section on this file for more details). The rdlHeader file includes all the declarations necessary to mix C programs with rule modules. For example, *relations* is a type defined in this file. Variables of this type must be declared to pass INPUT relations to modules and to obtain results from modules. This is the role of the liens variable. This variable should contain the list of INPUT relations needed by the Pcc2 module at the time of the call. The Pcc2 module will assign its result to this variable by adding to this list, the list of output relations that it has produced. The sql_connection and sql_deconnection procedures allow to establish and relinquish contact with the DBMS.

7.4. Debugging Programs

For the time being, it is only possible to debug programs by tracing its execution. To trace the execution of a rule program, the -t option should appear on the application command line. Consider the following compile command which compiles a module into an executable program.

```
rdlC mod1.r
```

To obtain a trace of the execution at mod1 run-time, the mod1 application should be run with the -t option as follows.

```
mod1 -t
```

The option will cause the inference engine to display (for each cycle) the list of pertinent rules; the rule that it chose to fire; for each element in the action part of the rule, the SQL query used to select those tuples that qualify the condition part of the rule; the number of tuples which qualify the condition; and the effect on the relations appearing in the action part of the rule.

The list of pertinent rules are those rules which are considered by the engine as possible candidates for firing. The rule that it will select for firing depends on the control string. For each relation appearing in the action part of the rule, the number of tuples that were present in the relation before firing are given along with the number of tuples present in the relation after firing the rule. A change in cardinality implies that the rule has had an effect on the relations in the action part of the rule.

If the module is implemented as a procedure, the -t option from the command line has to be passed to the module at the time of the call to obtain trace information from the inference engine. Consider the following command lines.

```
rdlC -pr mod1.r
Crdl main.c mod1.o
main -t
```

For the inference engine to deliver trace information when it will run module mod1, the -t option must be passed in the call to procedure mod1 from the main C program. Hence, the main C program must have the argc, argv parameters declared in the main procedure. These parameters must also be present in the call to mod1. This is shown in the following call to mod1.

```
outRels = mod1 ( argv, argc, inRels);
```

Reports can be run during a rule module execution for trace purposes. Consider the following rule.

```
r55 is
if P1(x) ...
then + depl_aff (x);
```

It might be interesting for the user to see the contents of the depl_aff relation each time that rule r55 fires and not only its final contents at the end of the program. To obtain this effect, the following post code can be added to the rule.

```
r55 is
if P1(x) ...
then + depl_aff (x);
{rep1();}
```

This procedure call will cause report rep1 to be executed each time rule r55 fires to display the contents of the depl_aff relation.

8. Running programs

Before running the application, the DBMS environment should be set. To set the environment, make sure that the DBMS files are present and issue a DBSTART command. If no error messages result from this command, the application is ready to run. Simply type in the name of the application to start it. If the module has been compiled as a program, the application will ask to user to enter the base name and password under which he wants to work. SYS is the system base name and is used to access all relations in the database.

8.1. The iml File

The following is the contents of the iml.h file which should be included in all applications which communicate with the DBMS. Important items have been flagged in bold. Non flagged items corresponds to less general operations which are detailed in the IML manual [Sabre89a].

```
/*          FICHIER de definition de types et de variables a inclure
      dans tout programme C utilisant l'interface IML
*/

/* Differentes valeurs du type attribut rendu par la primitive SQL_GETTUPLE  */
# define        ENTIER          0
# define        REEL            1
# define        TEXTE           2

#define         LGENTIER        4
#define         LGREEL          8
#define       MAXLIG            100
#define       LGMAXATT    255
#define       MAXA        40
#define       MAXPARAM    20
#define             DIM_TAB_REQ  3000 /*DIMENSION MAXI D'UNE REQUETE IML */

typedef char oct ;
typedef char typnomiml [MAXLIG] ;

struct texte    /* Type texte IML */
      {
        long Long ;
        char cont [LGMAXATT] ;
      } ;

/* Definition du type SCHEM_LISTE rendu par la primitive GET_SCH  */
struct schem_liste
        { long numatt;
          long tyat ;
          long lgattdef ;
        long tylisp ;
          struct texte t ;
          struct schem_liste *ps ;
        } ;

/* declaration des variables SABRE externes au programme d'application */
extern struct tsqlca  sqlca ;
extern long imlsqlcode ;
extern long imlnbtuple ;
extern long iml_nbatt ;
extern char * imllisp ;
extern long sabre_finrel;
extern tsql_attribut sql_attribut ;

/* declaration externes des primitives  IML  */
```

```
extern long sql_connection();
extern long sql_deconnection();
extern long sql_exec();
extern long sql_read();
extern long sql_usefile() ;
extern long sql_execfile() ;
extern long sql_gettuple();
extern struct schem_liste *sql_getschema();
```

Codes for the three basic domain types in the DBMS are given. These are integers, reals and text strings represented in French by ENTIER, REEL and TEXTE respectively. These codes are referred to by the tyat field of the schem_liste data structure. This field yields a number which is the base domain of the attribute. If the attribute is an ADT, the tylisp field is set to a value which is greater than 3. The t field is the name of the attribute. The ps field allows to link attributes together to obtain the list of attributes which comprise the relation.

Schema information is automatically obtained with a call to the sql_getschema function. This function takes two text arguments which are the base name and the relation name of the relation whose schema is to be obtained. For example, to obtain the schema of relation Toulouse.CTA, the following call should be issued.

```
sch = sql_getschema("Toulouse", "CTA");
```

If no such relation can be found in the database, the sch variable will be set to NULL.

To establish and relinquish contact with the DBMS, the sql_connection and sql_deconnection functions are used. The sql_connection function accepts two text parameters and returns a code that relates the status of the connection. A code value other than zero indicates a problem. The two connection parameters are the user's name and the pass word. There should be one such call per program. Consider the following example.

```
rep = sql_connection ("SYS", "");
```

This call connects the user SYS to the DBMS. There is no pass word associated to this user. To relinquish contact with the DBMS, the following procedure call is used.

```
rep = sql_deconnection();
```

This procedure takes no parameters.

The sql_exec procedure takes one parameter which is the sql statement that is to be processed. The number of tuples selected by the statement is assigned to a variable called imlnbtuple. Consider the following example.

```
rep = sql_exec ("select * from Toulouse.CTA;");
printf("The number of tuples is %d\n", imlnbtuple);
```

The first statement issues a select statement to the DBMS selecting all tuples from the Toulouse.CTA relation. The second statement displays the number of tuples that were selected by the call. The tuples can be read by an sql_read procedure call. See the Sabrina reference manual, [Sabrina1], for more information.

8.2. The rdlHeader File

This file must be included in programs which will interact with modules compiled as procedures. The interesting elements in this file have been highlighted in bold characters.

```
# include <stdio.h>
# define baseRel      1
# define deducedRel   2
# define outputRel    11
# define defRegle  0
# define defSeq    1
# define defBlock  2

/*=========================================*/
/* definition de types */

typedef struct {
  char *firstName;
  char *name;
  struct schem_liste *tlist;
  int typeRel;
  int  card;
} relation ;

typedef struct Srelations {
  relation *ptrelation;
  int projectionClause; /* for postplace only */
  relation *projRel; /* for postplace only */
  struct Srelations *next;
} relations ;
```

Three different types of relations are defined. These are base, deduced and output relations. These types are referenced by the typeRel field of the relation data structure. The relation data structure is the one used by the inference engine to manage relations during the execution of a rule program. The firstName field holds the name of the relation that was defined by the user in the rule module. The name field holds the name of the temporary relation which is the current state of the relation. This is done to overcome a problem in SQL naming conventions. Hence, a relation is identified by the user with the firstName field and its content is accessed with the name field. The tlist attribute holds the schema of the relation. The schem_list data structure has been detailed in the previous section on the iml header file. The card field holds the number of tuples in the relation. (This field is not valid for base relations. For these relations, the card field is assigned a zero value).

The relations data structure creates a list of relations. The ptrelation field is a pointer to a relation data structure. The next field links the elements in the list. Consider the following C main program segment.

```
relations *outRels, *inRels;
...
outRels = modl (argv, argc, inRels);
pt = outRels;
while (pt != NULL) {
  if (strcmp (pt->ptrelation->firstName, "aRelation") == 0) break;
  pt = pt->next;
}
if (pt == NULL) stopProgram();
```

```
...
rep = sql_exec ("select * from %s where a1 = 25", pt->ptrelation->name);
...
```

The output relations of module mod1 are obtained in the outRels variable. Both variables have been defined to be of type *relations*. The while loop allows to search for a relation called aRelation in this list. If the relation is not in the list, the program is halted. Otherwise, the relation produced by module mod1 is used in an SQL query. The string processing facility allows to replace the control sequence %s by the character string represented by the name field of the relation structure. This is the name of the temporary relation generated by the module to hold the tuples of relation aRelation.

9. Conclusion

The RDL/C language integrates a rule language and a procedural language (namely C). The kernel of the language is the RDL1 language. RDL/C is a proper subset of the RDL1 language.

For the development of database applications, the RDL/C language presents some interesting features compared to the C/SQL language. The first one is the expressive power of a rule compared to an SQL query. A rule might include several actions in its action part and is then equivalent to several SQL queries. A rule can be recursive and is equivalent to a C while loop over several SQL queries. Therefore, an RDL/C program is more concise than the equivalent C/SQL program. The second advantage over C/SQL programs is that the user is freed from the management of temporary relations. In RDL/C, these tasks are hidden from the user and managed by the inference engine. The resulting program is easier to develop and to maintain. Control in an RDL/C program might be written in two ways: In the language itself or with the control sub-language. This simulates standard control structures such as IF THEN ELSE and DO WHILE. Furthermore, RDL/C modules can be combined and called by C programs.

The data model is the relational one, (in the implementation we used as a testbed, it is extended with Abstract Data Types). Due to C and to SQL, the rule language is coupled with a query language. This is a useful feature for the development of applications.

10. References

[Gardarin89] G. Gardarin et al.: "Managing Complex Objects in an Extendible Relational DBMS", *Proc. of Int. Conf. on VLDB*, Amsterdam, Aug. 1989.

[Kiernan90] G. Kiernan, C. de Maindreville, E. Simon : "Making Deductive Database a Practical Technology: a step forward", *Proc. of SIGMOD 90*, Atlantic City NJ., June. 1990.

[Maindreville88] C. de Maindreville, E. Simon : "A Production Rule Based Approach to Deductive Databases", *Proc. of Int. Conf. on Data Engineering*, Los Angeles, Feb. 1988.

[Sabrina1] "Sabrina*SQL Volume 1 et 2", Reference Manual, 1989, Infosys, 15, rue A. France 92800 Puteaux, France.

[Sabrina2] "Sabrina*SQL-Object", Reference Manual, 1989, Infosys, 15, rue A. France 92800 Puteaux, France.

11. The BNF of the Language

```
programme       : includes module variables declrel reports RULES rules controls inits wrapup
                  END MODULE
                | error
                ;
includes        : include
                |
                ;
include         : INCLUDE CSTRING
                | include INCLUDE CSTRING
                ;
module          : MODULE ID ';'
                ;
declrel         : inputrels baserels deducedrels outputrels
                ;
inputrels       : INPUT declrels
                |
                ;
baserels        : BASE declrels
                |
                ;
deducedrels     : DEDUCED declrels
                |
                ;
outputrels      : OUTPUT  declrels
                |
                ;
declrels        : relat ';'
                | declrels relat ';'
                ;
relat           : rela  '(' schema ')'
                | rela LIKE rela
                ;
schema          : ID domaine
                | schema ',' ID domaine
                ;
reports         : REPORT reps
                |
                ;
reps            : report
                | reps report
                ;
report          : ID ON rela IS title header linesperpage format ';'
                | error
                ;
format          : FORMAT CSTRING
                | USE ID
                | USE ID '(' repparms ')'
                ;
repparms        : repparm
                | repparms ',' repparm
                ;
repparm         : ID
                | constbase
                ;
title           : TITLE CSTRING
                | /* no title */
                ;
header          : HEADER CSTRING
                | /* no header */
                ;
linesperpage    : LINESPPAGE INTNUMBER
                | /* no lines per page */
                ;
variables       : VAR listeVariableDecl
                |
                ;
listeVariableDecl : variableDecl
                  | listeVariableDecl variableDecl
```

```
                    ;
variableDecl        : externe domaine  listeVariables ';'
                    | error
                    ;
externe             : EXTERN
                    |
                    ;
listeVariables      : varname
                    | listeVariables ',' varname
                    ;
varname             : ID
                    | LETTRE
                    ;
inits               : INIT ccodee
                    |
                    ;
wrapup              : WRAPUP ccodee
                    |
                    ;
rules               : rule
                    | rules rule
                    ;
rule                : ID
                      IS ccode IF
                      declvars
                       conditions
                      alors
                      actions  ';' ccode
                    | error                          ';'
                    ;
alors               : THEN
                    | THENONCE
                    ;
ccode               : ccodee
                    |
                    ;
ccodee              : CCODE
                    | ccodee CCODE
                    ;
declvars            : rela '(' LETTRE ')'
                    | declvars  AND rela '(' LETTRE ')'
                    | /* empty */
                    ;
rela                : ID
                    | ID '.' ID
                    ;
conditions          : '(' condition ')'
                    | /* emptly condition*/
                    ;
condition           :  '(' condition ')'
                    | condition AND condition
                    | condition OR condition
                    | NOT condition
                    | cond
                    ;
cond                : compsimple
                    | formuquant
                    | PRED fonction
                    ;
compsimple          : compscal
                    | appinterval
                    | testvalnulle
                    | comptextfloue
                    ;
compscal            : expression relop expression
                    ;
relop               : EQ
                    | LT
                    | LE
                    | NE
                    | GT
                    | GE
```

```
                    ;
appinterval         : expression BETWEEN expression AND expression
                    | expression NOT BETWEEN expression AND expression
                    ;
testvalnulle        : expression IS NULL
                    | expression IS NOT NULL
                    ;
comptextfloue       : expression LIKE patron
                    | expression LIKE patron ESCAPE carescape
                    | expression NOT LIKE patron
                    | expression NOT LIKE patron ESCAPE carescape
                    ;
patron              : STRING
                    ;
carescape           : STRING
                    ;
formuquant          : EXISTS listevarquant
                    | EXISTS listevarquant '(' condition ')'
                    | FOREACH listevarquant '(' condition ')'
                    ;
listevarquant       : LETTRE IN rela
                    | listevarquant ',' LETTRE IN rela
                    ;
actions             :  projections
                    |  actions projections
                    ;
projections         : plusmoins rela  '(' projliste ')'
                    |  varname EQ expression
                    |  extcall
                    ;
extcall             : ID '(' extparms ')'
                    ;
extparms            : expression
                    | extparms ',' expression
                    ;
plusmoins           : '+'
                    | '-'
                    | PLUSPLUS
                    ;
projliste           : projection
                    | LETTRE
                    ;
projection          : ID EQ expression
                    | projection ',' ID EQ expression
                    ;
expression          :  '(' expression ')'
                    | expression '+' expression
                    | expression '-' expression
                    | expression '*' expression
                    | expression '/' expression
                    | expression MOD expression
                    | expression DIV expression

                    | '-' expression %prec UMINUS
                    | terme
                    ;
terme               : constante
                    | attribut
                    | fonction
                    ;
attribut            : ID
                    | LETTRE
                    | LETTRE '.' ID
                    ;
domaine             : dombase
                    | DOM
                    ;
dombase             : INTEGER
                    | REAL
                    | CHAR
                    ;
constante           : USER
```

```
                    | constbase
                    | constbase '[' domaine ']'
                    ;
constbase           : STRING
                    | INTNUMBER
                    | REALNUMBER
                    ;
fonction            : ID '(' listearguments ')'
                    | ID '(' ')'
                    ;
listearguments      : expression
                    | listearguments ',' expression
                    ;
controls            : CONTROL controlargs ';'
                    |
                    ;
controlargs         : BLOCK             '(' bsargs ')'
                    | SEQ               '(' bsargs ')'
                    ;
bsargs              : controlargs
                    | ID
                    | bsargs ',' controlargs
                    | bsargs ',' ID
                    ;
```

12. Error Messages

Error messages are supplied by the compiler at compile-time and by the application at run-time. At compile time, errors might occur in pure RDL/C statements or in C statements. Run-time errors can arise from different sources. Some errors are detected by the application. Among those that are not detected are of the type *division by zero* and the like. The user will get a *segmentation fault* message which is not particularly helpful. Among the errors that are detected at run-time by the application are schema errors for base and derived relations or errors arising in the SQL statements issued to solve rules. These error messages are discussed in the following sections.

12.1. Compile-Time Errors

The compiler has two phases. The first phase translates the rule source program into a C source program. Errors can be detected in this phase. If no errors are detected, the C source program is compiled by the C compiler. Errors in C statements appearing in the RDL/C source file might be detected in this phase. Error messages in this phase are usually self explanatory. Consider module search_District given as an example module in the sample application of this manual. An error can be created by an unknown relation called unKnown in the output declaration section. Since a relation is declared to be like relation unKnown, relation unKnown must have been declared for the program to be correct. The error message that the compiler will produce during the first phase is the following one.

```
*** ERROR The schema of unKnown is not defined
*** ERROR line 23 - makeSchemeLike : unKnown
*** ERROR line 23 - programme : unKnown
1 error detected
```

The first message gives an explicit indication as to the nature of the error. Such a message is printed provided that the error is not a pure syntax error, in which case there in no such message. The two following messages are always present. They give the line number where the error was detected, the name of grammar rule the compiler was trying to reduce the expression to (see the BNF of the language) followed by a colon and the token that caused the syntax error. This last information is only useful if the error is a pure syntax error.

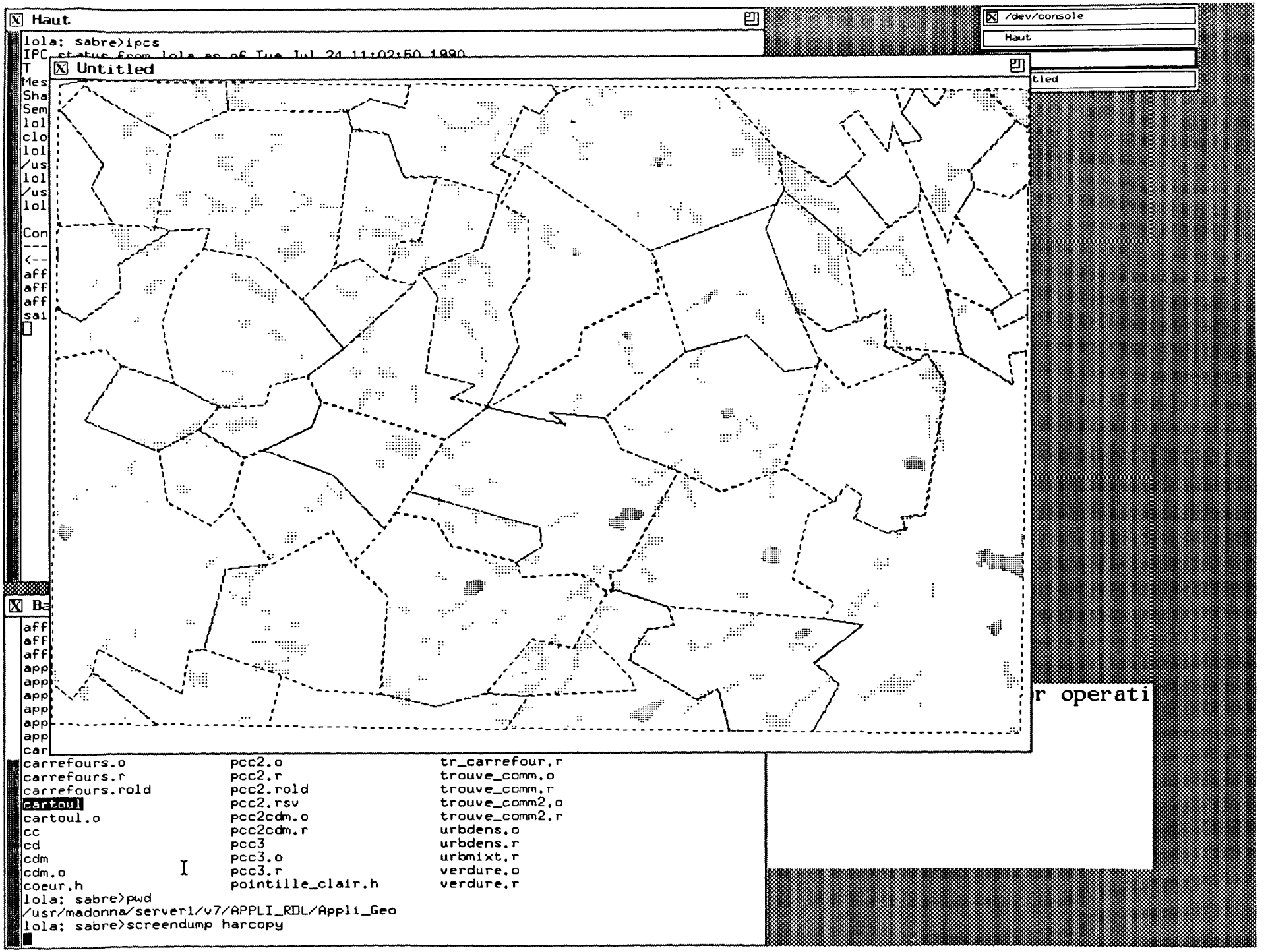
Errors produced by the C compiler in the second phase of evaluation refer to C code in the body of the rule source program. A compiler option allows the C compiler to refer to the line number in the rule source program where the error was detected.

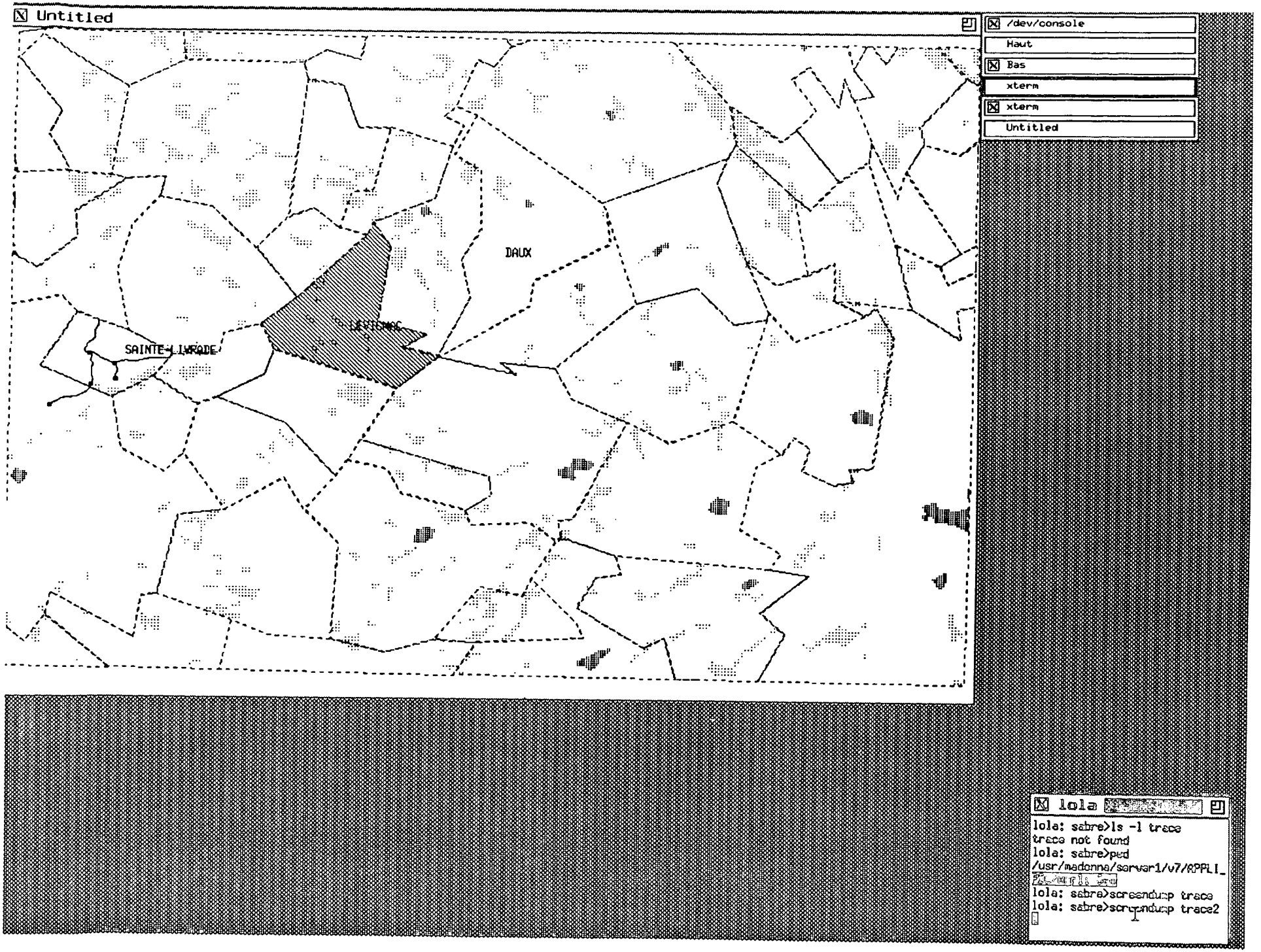
12.2. Run-Time Errors

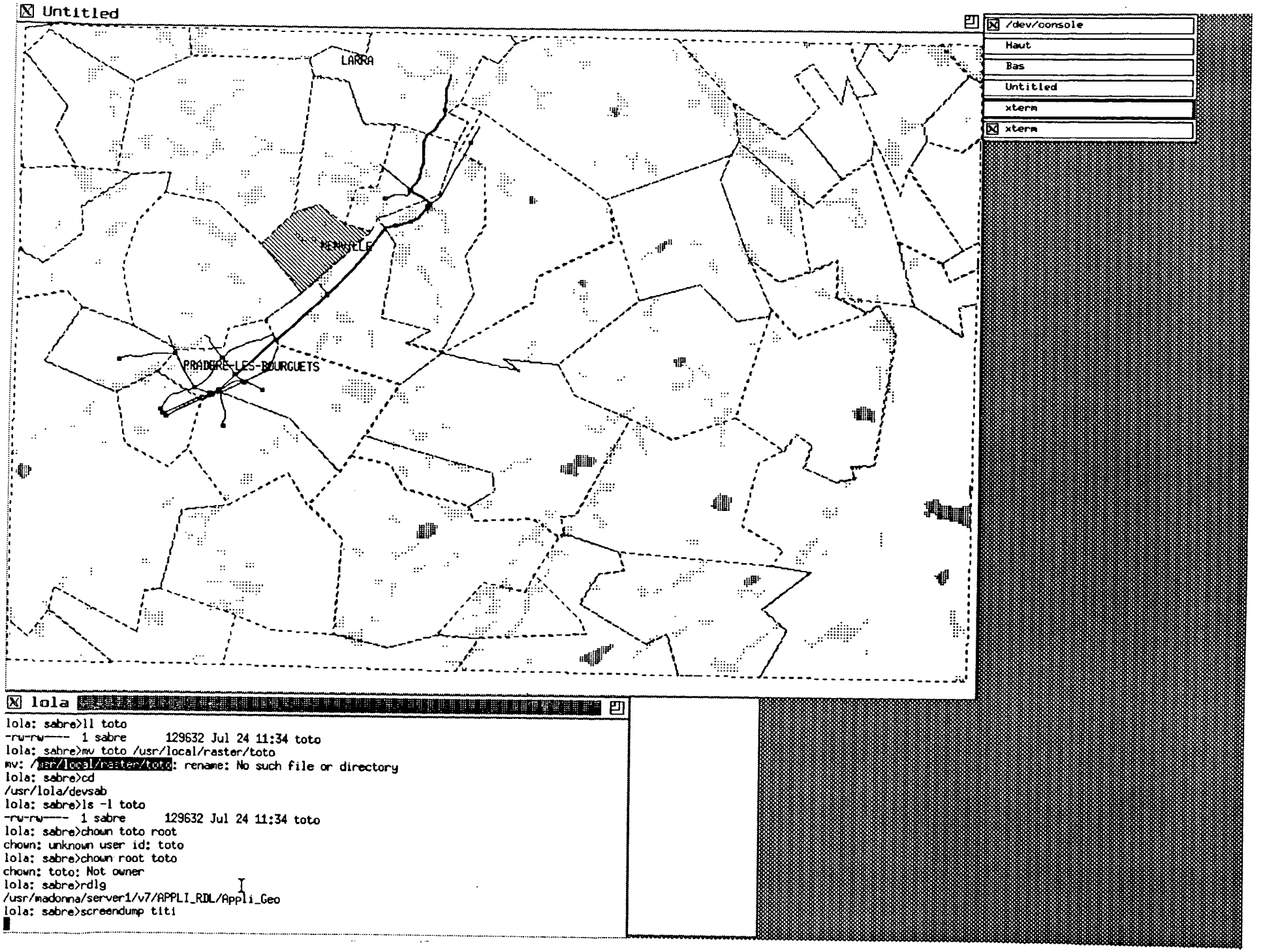
Some run-time errors are detected by the application. Consider one more time, module search_District given in the sample application section. A run-time error can be provoked by changing the function xyinpoly to xyinpoly1. The ADT function xyinpoly1 is not known to the DBMS system are will provoke a run-time error when the module tries to run the SQL query to solve the

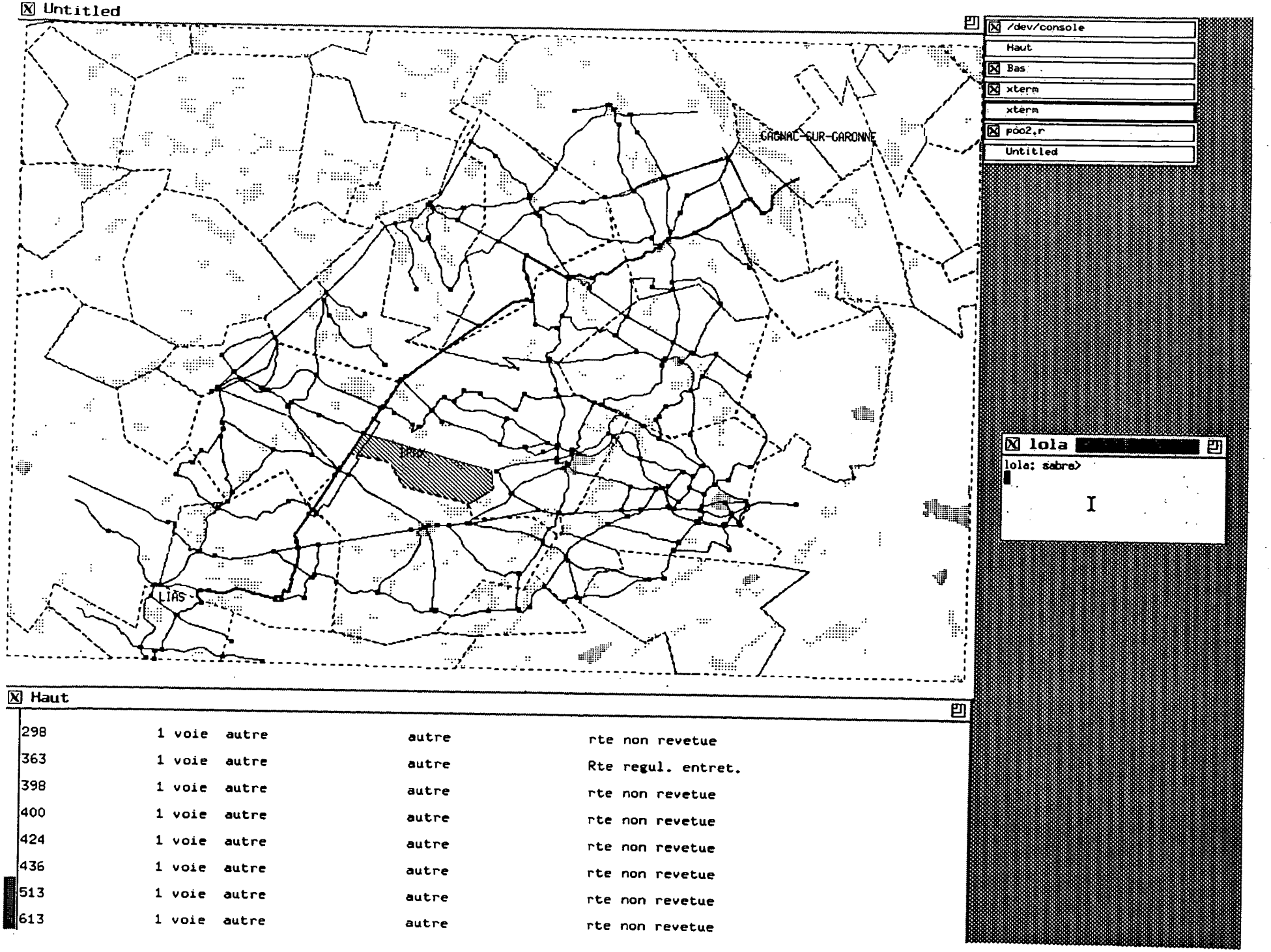
rule. First, an error is reported. -72 is the SQL code which corresponds to a function call that cannot be solved by the DBMS. This error code can also be detected in the application with the imlsqlcode variable, after the module call. (See the SQL manual for more detail on these codes). The message signals that the error was found in rule r2 in module search_District. The text of the SQL query that provoked the error is then given. This is an SQL query generated by the compiler to solve the rule. The function xyinpoly1 appears in the body of the query. In all cases, the schemas of the relations involved in the module are given. The District relation has not been assigned a relation, so it has no schema at the time of the error. The relation comm_inter has the temporary relation L1 assigned to it. The schema of L1 is given. For each attribute in the list, the name of the attribute comes first, followed by the base type of the attribute. The ADT type is given before the comma. Values of 1 to 3 correspond to integers, reals and texts respectively; all other values correspond to ADT and can be found in the SCHEMA.DEFLISP relation of the DBMS.

```
ERREUR -72 sur regle r2 dans search_District.

REQUETE SQL PROVOQUANT L'ERREUR
═══════════════════════════════

SELECT x.INSEE, x.ARRD, x.REGION, x.CODE, x.NAME, x.POPU, x.X, x.Y, x.XMIN,
x.YMIN, x.
XMAX, x.YMAX, x.GEOM FROM L1 as x WHERE xyinpoly1( 4357 , 3236
,geom_to_poly( x.X
MIN,x.YMIN,x.GEOM)) = 'T' ;

Schemas des relations dans l'environnement
══════════════════════════════════════════

District -> District ()
comm_inter -> L1 (INSEE TEXTE, 3, ARRD ENTIER, 1, REGION ENTIER, 1, CODE
ENTIER, 1, NAME TEXTE, 3, POPU ENTIER, 1, X ENTIER, 1, Y ENTIER, 1, XMIN
ENTIER, 1, YMIN ENTIER, 1, XMAX ENTIER, 1, YMAX ENTIER, 1, GEOM TEXTE, 14)
TOULOUSE.DISTRICT -> TOULOUSE.DISTRICT (INSEE TEXTE, 3, ARRD ENTIER, 1, REGION
ENTIER,
1, CODE ENTIER, 1, NAME TEXTE, 3, POPU ENTIER, 1, X ENTIER, 1, Y ENTIER, 1,
XMIN ENTIER, 1, YMIN ENTIER, 1, XMAX ENTIER, 1, YMAX ENTIER, 1, GEOM TEXTE, 14)
```


Haut
Untitled
/dev/console
Haut
lola: sabre>ipcs
carrefours.o
carrefours.r
carrefours.rold
cartoul
cartoul.o
cc
cd
cdm
cdm.o
coeur.h
pcc2.o
pcc2.r
pcc2.rold
pcc2.rsv
pcc2cdm.o
pcc2cdm.r
pcc3
pcc3.o
pcc3.r
pointille_clair.h
tr_carrefour.r
trouve_comm.o
trouve_comm.r
trouve_comm2.o
trouve_comm2.r
urbdens.o
urbdens.r
urbmixt.r
verdure.o
verdure.r
lola: sabre>pwd
/usr/madonna/server1/v7/APPLI_RDL/Appli_Geo
lola: sabre>screendump harcopy

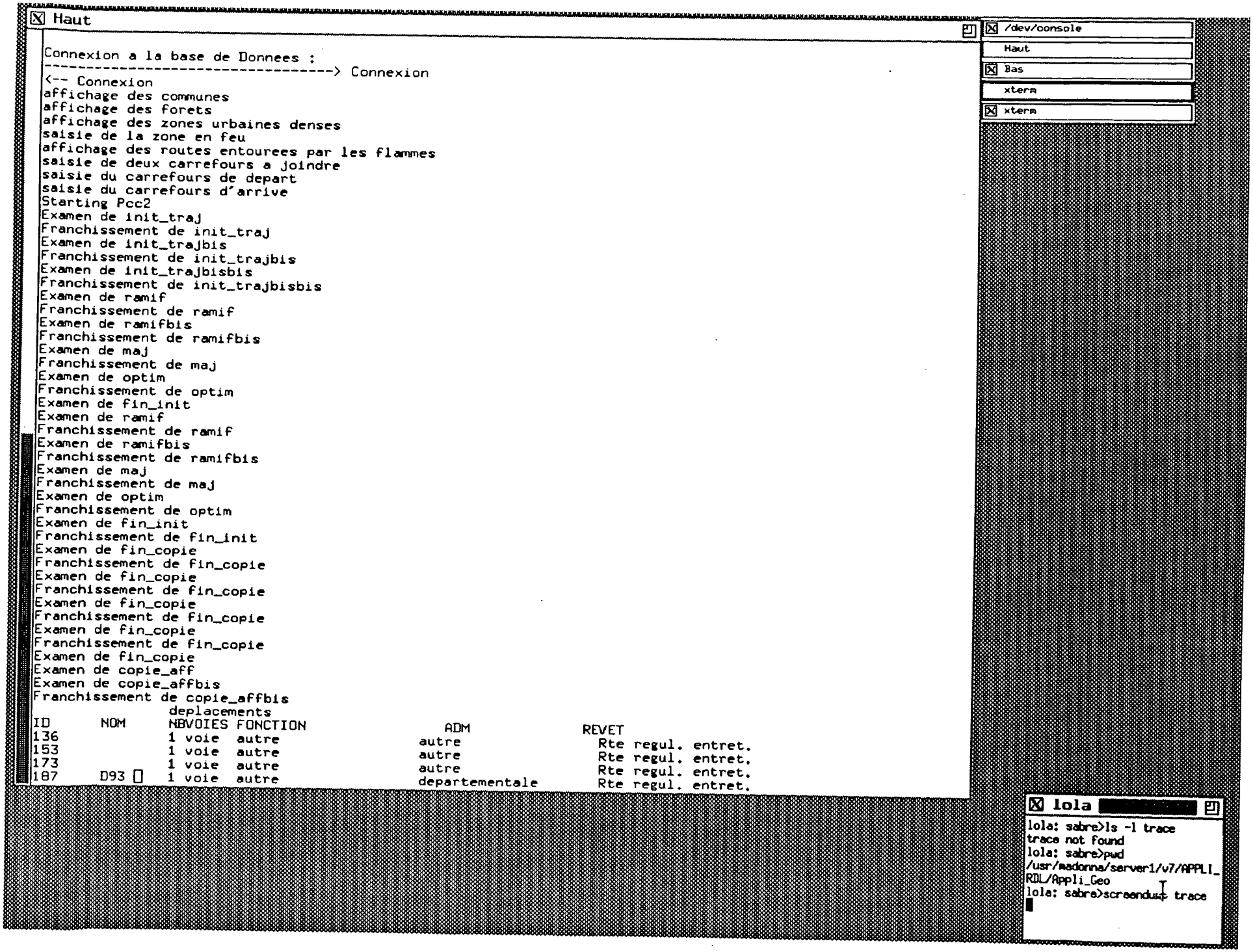

Untitled
/dev/console
Haut
Bas
xterm
xterm
Untitled
DAUX
SAINTE-LIVRADE
lola
lola: sabre>ls -l trace
trace not found
lola: sabre>pwd
/usr/madonna/server1/v7/APPLI_
lola: sabre>screendump trace
lola: sabre>screendump trace2


Untitled
LARRA
MEMAILLE
PRADERE-LES-BOURGUETS
/dev/oonsole
Haut
Bas
Untitled
xterm
xterm
lola
lola: sabre>ll toto
-rw-rw---- 1 sabre 129632 Jul 24 11:34 toto
lola: sabre>mv toto /usr/local/raster/toto
mv: /usr/local/raster/toto: rename: No such file or directory
lola: sabre>cd
/usr/lola/devsab
lola: sabre>ls -l toto
-rw-rw---- 1 sabre 129632 Jul 24 11:34 toto
lola: sabre>chown toto root
chown: unknown user id: toto
lola: sabre>chown root toto
chown: toto: Not owner
lola: sabre>rdlg
/usr/madonna/server1/v7/APPLI_RDL/Appli_Geo
lola: sabre>screendump titi



Haut

298	1 voie	autre	autre	rte non revetue
363	1 voie	autre	autre	Rte regul. entret.
398	1 voie	autre	autre	rte non revetue
400	1 voie	autre	autre	rte non revetue
424	1 voie	autre	autre	rte non revetue
436	1 voie	autre	autre	rte non revetue
513	1 voie	autre	autre	rte non revetue
613	1 voie	autre	autre	rte non revetue


Haut
Connexion a la base de Donnees :
----------> Connexion
<-- Connexion
affichage des communes
affichage des forets
affichage des zones urbaines denses
saisie de la zone en feu
affichage des routes entourees par les flammes
saisie de deux carrefours a joindre
saisie du carrefours de depart
saisie du carrefours d'arrive
Starting Pcc2
Examen de init_traj
Franchissement de init_traj
Examen de init_trajbis
Franchissement de init_trajbis
Examen de init_trajbisbis
Franchissement de init_trajbisbis
Examen de ramif
Franchissement de ramif
Examen de ramifbis
Franchissement de ramifbis
Examen de maj
Franchissement de maj
Examen de optim
Franchissement de optim
Examen de fin_init
Examen de ramif
Franchissement de ramif
Examen de ramifbis
Franchissement de ramifbis
Examen de maj
Franchissement de maj
Examen de optim
Franchissement de optim
Examen de fin_init
Franchissement de fin_init
Examen de fin_copie
Franchissement de fin_copie
Examen de fin_copie
Franchissement de fin_copie
Examen de fin_copie
Franchissement de fin_copie
Examen de fin_copie
Franchissement de fin_copie
Examen de fin_copie
Examen de copie_aff
Examen de copie_affbis
Franchissement de copie_affbis
ID NOM NBVOIES FONCTION ADM REVET
deplacements
136 1 voie autre autre Rte regul. entret.
153 1 voie autre autre Rte regul. entret.
173 1 voie autre autre Rte regul. entret.
187 D93 1 voie autre departementale Rte regul. entret.
/dev/console
Haut
Bas
xterm
xterm
lola
lola: sabre>ls -l trace
trace not found
lola: sabre>pwd
/usr/madonna/server1/v7/APPLI_RDL/Appli_Geo
lola: sabre>screendump trace